МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”
ІНСТИТУТ ТЕЛЕКОМУНІКАЦІЙНИХ СИСТЕМ

Л.С. Глоба

ПІДРУЧНИК

“РОЗРОБКА ІНФОРМАЦІЙНИХ РЕСУРСІВ ТА СИСТЕМ”

(Том 1: «Розподілені системи», «Розподілені системи. Поняття розподіленого середовища», «Зв’язок», «Процеси», «Іменування», «Синхронизація»)

для студентів спеціальностей
8.092401
«Телекомунікаційні системи та мережі»

8.092402
«Інформаційні мережі зв’язку»

Київ
„Політехніка”
2013

ЗМІСТ

ПЕРЕДМОВА

Підручник «Розробка інформаційних ресурсів та систем» призначено для інженерів, аспірантів, а також студентів-магістрів спеціальностей 8.092401 «Телекомунікаційні системи та мережі» та 8.092402 «Інформаційні мережі зв'язку». В ньому викладено методологічні основи побудови розподілених інформаційно-телекомунікаційних систем та середовищ, зокрема Internet-базованих, принципи утворення інформаційно-обчислювального середовища корпоративних систем прикладного призначення, а також підходи до просування інформаційних ресурсів компаній у пошукових машинах. Зростання обсягів інформатизації в усіх галузях життєдіяльності людини і суспільства потребує підготовки саме таких спеціалістів, здатних до самостійного опанування різноманітних аспектів роботи у сфері телекомунікаційних та комп'ютерних технологій в умовах необхідності у стислі терміни ефективно опрацьовувати значні обсяги інформації.

Системи розподіленої обробки даних у Intranet-мережі належать до найбільш прогресивних форм організації програмно-технічних засобів у вигляді продуктивного інформаційно-телекомунікаційного середовища, в їх основі лежать технології паралельних і «хмарних» обчислень. Їх проектування і реалізація вимагають умов роботи, за яких користувачі потребують доступу до всіх файлів, що зберігаються у вузлах інформаційно-телекомунікаційної мережі. Ефективність доступу користувачів значною мірою залежить від організації інформаційно-обчислювального середовища, у тому числі з використанням мережі Internet.

Оскільки більшість корпоративних інформаційно-телекомунікаційних систем вирізняється з-поміж інших як апаратним, так і програмним забезпеченням, організувати колективну роботу користувачів у оперативному режимі досить складно через суттєву неструктурованість інформації. Виходячи з цього, слід зазначити, що оптимізація ефективності роботи корпоративних систем стає нагальною проблемою, вирішення якої можливе, якщо одночасно використовувати низку сучасних технологій зв'язку та інформаційних технологій.

Вперше такий підхід було запропоновано в роботах Э.Таненбаума [1,2], де було чітко окреслено основні складові сучасних розподілених систем. Даний підхід став класичним теоретичним результатом розвитку інформаційно-телекомунікаційних технологій та систем, його було поширено в роботах багатьох науковців [3, 4, 7, 10, 11, 12, 21, 28, 32, 33, 76, 80-98]. Виходячи з того, що багато сучасних наукових підходів для створення ефективних глобальних систем та технологій [15, 19, 26, 59, 69, 72] використовують структуру, протоколи, методи, алгоритми побудови розподілених систем, дуже чітко представлені в роботах Э.Таненбаума, в підручнику автором взято за основу структуру даного підходу, застосовано його як базовий зі збереженням основних структурних складових, в значній мірі використано матеріали, наведені в його роботах.

Перший том складається із шести розділів, у яких, поряд із досить повним висвітленням основ Internet-технологій, розглянуто ґрунтовно телекомунікаційні технології, інформаційні технології, які підтримують роботу в глобальному середовищі, приклади сучасних Internet-базованих систем. Відповідно розглянуто принципи створення, організації доступу та збереження інформаційних і програмних ресурсів у інфраструктурі територіально-розподілених систем та середовищ, архітектурні рішення, які визначають ефективність їх використання. Особливу увагу приділено телекомунікаційним технологіям, які забезпечують безпосередній зв’язок між апаратними засобами, апаратними засобами і людиною, між людиною та людиною через фіксовані та безпроводові мережі зв’язку, в їх основу покладено IP-протоколи; поняттю потоків та процесів; виконанню багатопотокових процесів у глобальному середовищі; підтримці глобальної системи іменування в середовищі Internet. Потреба у сучасних технологіях, які забезпечують ефективний доступ до інформаційних та обчислювальних ресурсів у Internet, загострює питання вдосконалення методів синхронізації процесів інформаційного обміну, які й висвітлено у підручнику.

Подальше все більш глибоке проникнення в усі сфери діяльності людини інформаційно-телекомунікаційних технологій вимагає істотного підвищення ефективності процесів інформаційного обміну, що забезпечується як за рахунок удосконалення апаратної інфраструктури, каналів зв’язку, прото-

колів підтримки їх функціонування, які мають певні фізичні обмеження за своєю природою, так і за рахунок упровадження нових програмних рішень, таких як програмовані апаратні засоби, нові архітектурні рішення щодо інтеграції різноманітних ресурсів, нові інформаційні технології, інтерфейси та протоколи. Все це визначає напрям інтеграцію інформаційних і телекомунікаційних технологій як магістральний напрям у цій галузі знань.

Інтеграція знань у сфері інформаційно-телекомунікаційних технологій забезпечує можливість утворення єдиного корпоративного простору певної організації, групи людей, окремої особи, розподілених середовищ колективної роботи, ця проблематика проходить через усі розділи підручника.

Вивчаючи матеріал, поданий у підручнику, студенти мають усвідомити математичні й технічні принципи побудови розподілених в інформаційно-телекомунікаційній мережі систем, набути вмінь грамотно проектувати складні розподілені у глобальній мережі системи і кваліфіковано формувати завдання на їх розробку.

Підручник передбачає ґрунтовне ознайомлення студентів із сучасним рівнем розвитку інформаційно-телекомунікаційних технологій.

Методика викладення навчального матеріалу, присвяченого питанням проектування та ефективного функціонування сучасних розподілених в Intranet-мережі систем, відповідає завданням і вимогам, обумовленим якісно новим рівнем інформатизації суспільства, у зв’язку з чим суттєво збільшено розділи, стосовно програмних технологій побудови Intranet-мереж.

Зміст підручника ґрунтується на досвіді викладання відповідних курсів на кафедрі «Інформаційно-телекомунікаційних мереж» (ІТМ) Інституту телекомунікаційних систем (ІТС) Національного технічного університету України «Київський політехнічний інститут» (НТУУ «КПІ»). Певні розділи посібника становлять змістовну основу віртуальної лабораторії комп’ютерних мереж, яка входить до інформаційного забезпечення навчально-дослідницької діяльності учнівської молоді за програмами Малої Академії наук України.

Автор висловлює щиру подяку академіку Національної академії наук України, проректору з наукової роботи, директору ІТС НТУУ «КПІ» профе-

сору М.Ю. Ільченко, рецензентам підручника декану механіко-математичного факультету Харківського Національного університету ім. В. Н. Каразіна, завідувачу кафедри теоретичної та прикладної інформатики професору Г. М. Жолткевичу та професору Національного університету України «Львівська політехніка» М. М. Климашу, директору Навчально-наукового інституту телекомунікацій та інформатизації Державного університету інформаційно-комунікаційних технологій, професору Л. Н. Беркман за уважний розгляд рукопису і цінні рекомендації, що допомогло суттєво покращити його змістову частину, а також секретареві кафедри ІТМ НТУУ «КПІ» О. В. Гетьман, аспірантці К. О. Єрмаковій за допомогу під час оформлення підручника та студентам груп ТІ-51м, ТІ-52м, ТС-51м, ТС-52м за активну участь у формуванні підручника.

Відгуки про підручник просимо надсилати за адресою:

03056, м. Київ, пров. Індустріальний, 2, корпус 30, Інститут телекомунікаційних систем, професору Л. С. Глобі.

ВСТУП

Можливості сучасних інформаційно-телекомунікаційних систем залежать від рівня науково-технічних розробок у сфері програмно-технічних засобів та систем збору, передачі, доступу та аналізу інформації, поєднуючи таким чином програмно-апаратні засоби і технології зв'язку, засоби обчислювальної техніки, програмні технології та системи тощо.

З огляду на це у навчальні плани підготовки магістрів входить дисципліна «Розробка інформаційних ресурсів та систем» циклу професійної та практичної підготовки, яка забезпечує підготовку студентів-магістрів напряму «Телекомунікації» за спеціальностями 8.092401 «Телекомунікаційні системи та мережі» та 8.092402 «Інформаційні мережі зв'язку». Дисципліна є профільною у підготовці фахівців зі створення та експлуатації корпоративних інформаційно-телекомунікаційних систем і мереж, інформаційних та обчислювальних ресурсів операторів зв'язку й контент-провайдерів, платформ розгортання IP-телебачення.

Сучасні технології побудови систем та мереж зв'язку, їх стандарти і протоколи застосовують як телекомунікаційні технології, так і програмні технології, які, у свою чергу, все більше використовують IP-протоколи та Internet/Intranet-технології. Розподілені (або Internet-базовані) інформаційні системи за рівнями складаються з апаратної інфраструктури (апаратура й обчислювальна техніка), рівня мережної інфраструктури, рівня мережних операційних систем, служб мережних операційних систем, рівня служб програмного забезпечення проміжного рівня та рівня розподілених прикладних програм. На їх основі утворюються інформаційно-телекомунікаційні середовища колективного користування, різноманітні прикладні інформаційні системи, які працюють у глобальному середовищі, пошукові системи типу Google, засоби й технології доступу до віддалених інформаційних та обчислювальних ресурсів, системи публікації-підписки тощо.

Концепції проектування та експлуатації сучасних Internet-базованих інформаційних систем об'єднали досягнення багатьох галузей знань, таких як електроніка та обчислювальна техніка; телекомунікації (теорія зв'язку, мережні технології); програмні технології (інформатика, технології програ-

мування, теорія баз даних та знань); математика (теорія ймовірностей, теорія масового обслуговування, дискретна математика).

Знання у сфері створення та функціонування інформаційно-телекомунікаційних систем і середовищ потрібні для засвоєння дисциплін, які вивчають принципи побудови та роботу різноманітних платформ операторів зв'язку [39], контент-провайдерів, корпоративних інформаційних систем прикладного призначення, зокрема банківських платіжних систем, інформаційних порталів, засобів масової інформації, IP-телебачення [47], CRM-систем та аналогічних Internet-базованих інформаційних систем [42].

Дисципліна надає знання у сфері сучасних технологій розробки програмного забезпечення на основі платформ Unix та Windows, використовуючи групи протоколів TCP/IP, HTTP, SOAP та інших, інструментальні засоби і технології програмування провідних виробників; вивчає особливості розподілених інформаційно-телекомунікаційних систем, що визначають перелік програмних технологій та засобів їх розробки, а також такі особливості, як розподіленість архітектури технічних засобів, програмних функцій, даних, реалізація архітектури Client-Server на основі WWW.

Під час вивчення дисципліни головними завданнями є засвоєння основних принципів та методології побудови і проектування інформаційно-телекомунікаційних систем електронної обробки (EDP) та обміну даними (EDI) на основі Intranet-технологій; визначення основних видів інформаційних та обчислювальних ресурсів; засвоєння провідних методів створення інформаційних ресурсів; методів адміністрування розподілених інформаційно-телекомунікаційних систем, які функціонують у глобальному середовищі; методів організації інтерфейсу користувача в розподілених інформаційних системах; застосування сучасного інструментарію проектування архітектури програмного та інформаційного забезпечення Internet-базованих систем; проектування розподілених баз та сховищ даних, ресурсів будь-якого обсягу, технологій їх взаємодії.

Дисципліна «Розробка інформаційних ресурсів та систем» забезпечується інформаційними й телекомунікаційними технологіями, які є інтеграційним науково-технічним напрямом, що охоплює проблеми дослідження, проекту-

вання, взаємодії та доступу до інформаційних та обчислювальних ресурсів і сервісів у глобальному інформаційно-телекомунікаційному середовищі.

Проблематику створення глобального інформаційно-обчислювального середовища та отримання різноманітних сервісів у ньому поділяють на дві групи глибоко інтегрованих технологій: інформаційні технології [42,44.45,46] і технології зв'язку [39,40,41,43]. Інформаційні технології підтримують доступ до ресурсів та сервісів на верхніх рівнях моделі OSI, а технології зв'язку призначено для підтримки глобальної інфраструктури, яка забезпечує зберігання та доступ до всіх ресурсів мережі на нижніх рівнях моделі OSI.

1. РОЗПОДІЛЕНІ СИСТЕМИ

1.1. Історична довідка

Швидкість підвищення продуктивності, що спостерігалася в комп'ютерних технологіях в останні півстоліття, – значна. Розвиток пройшов від машин, що коштували 100 млн. дол. та виконували одну команду за секунду, до машин, які коштують 1 тис. дол. і виконують 10 млн. команд за секунду. Різниця у співвідношенні ціна/продуктивність досягла порядку 10^{12}. Темпи збільшення продуктивності розподілених систем і технології, за допомогою яких це було досягнуто, показано на рис. 1.1.

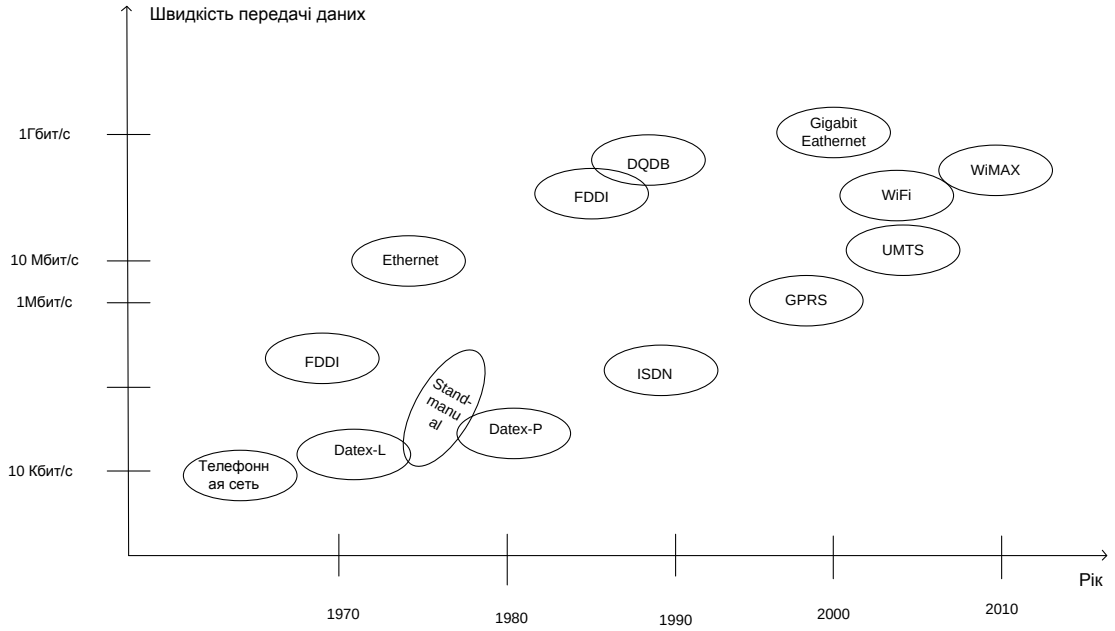


Рис. 1.1. Підвищення продуктивності в комп'ютерних мережах

Загальна тенденція відмови від централізації почала виникати із середини вісімдесятих років, коли американська фірма «Intel» запропонувала замість інтегрального модуля із твердою логікою розробити стандартний логічний блок, конкретне призначення якого можна сформулювати після його виготовлення, тобто створити програмовану інтегральну схему. Так з'явився мікропроцесор – багатофункціональний цифровий мікроелектронний модуль з програмованою логікою, який став революційним кроком в електроніці й техніці обробки інформації.

Важливим є й факт винаходу високошвидкісних комп'ютерних мереж. Локальні мережі (Local-Area Networks, LAN) стали з'єднувати сотні комп'ютерів, дозволяти їм обмінюватися невеликими порціями інформації за кілька мікросекунд. Великі масиви даних почали передаватися з машини на машину зі швидкістю від 10 до 1000 Мбіт/с.

Відмову від централізації зумовлено такими тенденціями розвитку.

В останні десятиліття XX століття продуктивність мікропроцесорів стрімко подвоїлася, а потужність керувальних чипів збільшувалася в чотири рази кожні три роки. Безупинно зростаюча продуктивність за умови зменшення цін і габаритів утворювала основу для того, що все більше комп'ютерів могли працювати зі складнішим програмним забезпеченням.

Локальні мережі даних стали економічною передумовою для того, щоб поєднати персональні комп'ютери. Впровадження Ethernet-технології в сімдесяті роки можна розглядати як першу технологію, яка дозволила побудувати розподілену систему, що працює під керуванням програмного забезпечення, яке надавало можливість доступу до різних апаратних і програмних ресурсів.

Слід звернути увагу на суттєвий прогрес в останні десятиліття у сфері програмних технологій. Введення в мовах програмування таких концепцій, як процедура, модуль та інтерфейс стало передумовою для створення основних механізмів розподілених систем, результатом чого було розроблено технології віддаленого виклику прикладних програм RPC (Remote Procedure Call) і об'єктно-орієнтоване проектування розподілених систем.

Швидке проникнення інформаційних технологій у комерцію, банківську справу, освіту, зв'язок і сферу послуг разом з потужністю комп'ютерів і об'ємом пристроїв зберігання даних, які постійно зростають, висуває суворі вимоги до мереж зв'язку.

Локальні мережі вже сформували глобальні мережі (Wide-Area Networks, WAN), які дозволяють мільйонам машин у всьому світі обмінюватися інформацією зі швидкостями, що варіюються від 64 кбіт/с до декількох гігабіт за секунду.

У результаті розвитку мережних технологій можна побудувати систему, що складається з низки комп'ютерів й інших пристроїв, з'єднаних високош-

видкісною мережею, яку називають інформаційною або розподіленою системою (distributed system), на відміну від попередніх централізованих (centralized systems) або однопроцесорних (single-processor systems) систем, тобто систем, які складалися з одного комп'ютера, його периферії й, можливо, декількох віддалених терміналів.

Досить багато підходів, характерних для розподілених інформаційних систем, є аналогічними для побудови апаратного забезпечення звичайних персональних комп'ютерів. Особливо це стосується багатопроцесорних материнських плат, багатоядерних процесорів і, найбільше, сучасних відеокарт. У відеокартах використання одного потужного спеціалізованого процесора виявилося значно дорожчим рішенням, ніж поєднання раніше створених, більш простих спеціалізованих процесорів. Використання групи спеціалізованих процесорів дозволило підвищити надійність апаратури, полегшити тепловідведення.

1.2. Базові терміни та визначення

Поняття «розподілена система» в літературних джерелах визначають по-різному. Історично у процесі розвитку апаратної та програмної складових розподілених систем формувалися різні розуміння їх як систем, що відображалося на їх визначеннях. Розуміння того, як має визначатися розподілена система, постійно уточнюється.

1. **Розподіленою системою** називають низку з'єднаних центральних процесорів (CPU), що працюють разом.

2. **Розподіленою системою** називають низку машин з нерозділеною пам'яттю.

Надалі розподілені системи потрібно розглядати в найбільш загальній формі. До них належать усі різновиди клієнт-серверних систем, а також мультипроцесорні системи, тобто такі, які складаються з вузлів та можуть бути однопроцесорними або мультипроцесорними.

3. **Розподіленою** називають систему з просторово розподіленими компонентами, які не використовують ніякої спільної пам'яті й не підлягають

децентралізованому адмініструванню. Для реалізації спільних цілей можлива кооперація компонентів. Якщо цими компонентами пропонуються послуги або використовуються запропоновані послуги, то виникає клієнт-серверна система, а у разі додаткового центрального службового посередництва – «торговельна» система (Trading-система).

Часто, визначаючи розподілену систему, на перше місце ставлять поділ її функцій між декількома комп'ютерами. За такого підходу розподіленою є будь-яка обчислювальна система, де обробку даних розділено між двома й більше комп'ютерами.

Розподілена система – це набір незалежних комп'ютерів, які користувач сприймає як єдину об'єднану систему.

У цьому визначенні є два однаково важливі моменти: стосовно апаратури – всі машини автономні; стосовно програмного забезпечення – користувачам надається у користування єдина система.

Основними завданнями розподіленої системи є організація ефективного доступу користувачів до інформаційних і програмних ресурсів та ефективна взаємодія як користувачів з ресурсами, так і різних видів ресурсів між собою.

1.2.1. Телекомунікаційні мережі, як елемент розподілених систем

Основною транспортною інфраструктурою розподілених систем є **телекомунікаційна мережа**, яка дозволяє виконувати інформаційний обмін в глобальному середовищі. Однією з важливих складових телекомунікаційної мережі є комп'ютерна мережа.

Комп'ютерні мережі належать до розподілених (або децентралізованих) обчислювальних систем. В обчислювальних мережах програмні й апаратні зв'язки є слабшими, а автономність розрахункових блоків простежується найбільшою мірою – основними елементами мережі є стандартні комп'ютери, що не мають ні спільних блоків пам'яті, ні спільних периферійних пристроїв. Зв'язок між комп'ютерами здійснюється за допомогою спеціальних периферійних пристроїв – мережних адаптерів, сполучених каналами зв'язку, які мають порівняно велику протяжність. Кожний комп'ютер працює під керу-

ванням власної операційної системи, а деякої «спільної» операційної системи, яка розподіляє роботу між комп'ютерами мережі, немає. Взаємодія між комп'ютерами мережі відбувається за рахунок передачі повідомлень через мережні адаптери і канали зв'язку. За допомогою цих повідомлень один комп'ютер запитує дозвіл на доступ до локальних ресурсів іншого комп'ютера. Такими ресурсами можуть бути як дані, що зберігаються на диску, так і різноманітні периферійні пристрої – принтери, модеми, факсимільні апарати та ін. Поділ локальних ресурсів кожного комп'ютера між усіма користувачами мережі є основною метою створення обчислювальної мережі.

На комп'ютерах, ресурси яких мають бути доступними всім користувачам мережі, необхідно встановити модулі, що постійно перебуватимуть у режимі очікування запитів, які надходять мережею від інших комп'ютерів. Зазвичай такі модулі називають **програмними серверами** (server), тому що їх головне завдання – обслуговувати (to serve) запити на доступ до ресурсів свого комп'ютера. На комп'ютерах, користувачі яких хочуть отримувати доступ до ресурсів інших комп'ютерів, також потрібно додати до операційної системи деякі спеціальні програмні модулі, що мають виконувати запити на доступ до віддалених ресурсів і передавати їх мережею на потрібний комп'ютер. Такі модулі зазвичай називають **програмними клієнтами** (client). Власне мережні адаптери й канали зв'язку вирішують у мережі достатньо просте завдання: передають повідомлення із запитами і відповідями від одного комп'ютера до другого, а основну роботу з організації спільного використання ресурсів виконують клієнтські й серверні частини операційних систем. Пара модулів клієнт–сервер забезпечує спільний доступ користувачів до визначеного типу ресурсів, наприклад до файлів, у цьому разі вважають, що користувач застосовує файлову службу. Зазвичай мережна операційна система підтримує декілька видів мережних служб для своїх користувачів: файлову службу, службу друку, службу електронної пошти, службу віддаленого доступу та ін.

У технічній літературі англомовний термін «service» зазвичай перекладають як «служба», «сервіс» і використовують як синоніми. Водночас, деякі спеціалісти розрізняють термін «служба», з одного боку, й терміни «сервіс» і «послуга», з другого. Під службою розуміють мережний компонент, що реа-

лізує деякий набір послуг, а під сервісом – перелік послуг, який надає ця служба. Таким чином, сервіс – це інтерфейс між споживачем послуг і їх постачальником (службою).

Терміни «клієнт» і «сервер» використовують для позначення не тільки програмних модулів, але й комп'ютерів або пристроїв, під'єднаних до мережі. Якщо комп'ютер надає свої ресурси іншим комп'ютерам мережі, то його називають сервером, а якщо він їх споживає – клієнтом. Іноді один комп'ютер може одночасно перебувати і в ролі сервера, і в ролі клієнта. Мережні служби завжди є розподіленими програмами загальносистемного призначення, які складаються з декількох частин, що взаємодіють, причому кожна частина переважно реалізується на окремому комп'ютері мережі, за допомогою її можуть виконуватися і розподілені програми користувачів. Розподілена прикладна програма також містить декілька частин, кожна з яких виконує якусь визначену закінчену роботу з вирішення прикладного завдання. Наприклад, одна частина прикладної програми, яку користувач виконує на власному комп'ютері, може підтримувати спеціалізований графічний інтерфейс; друга – працювати на потужному віддаленому комп'ютері й виконувати статистичну обробку введених користувачем даних; а третя – заносити отримані результати в базу даних на комп'ютер з установленою стандартною СУБД.

Розподілені прикладні програми повною мірою використовують потенційні можливості розподіленої обробки, надані обчислювальною мережею, тому їх називають мережними прикладними програмами.

Варто відзначити, що не кожна прикладна програма, яка виконується у мережі, є мережною. Велика кількість популярних прикладних програм не є розподіленими і повністю виконуються на одному комп'ютері в мережі. Проте й такі прикладні програми можуть використовувати переваги мережі за рахунок вмонтованих в операційну систему мережних служб.

Більшість прикладних програм, які використовувалися у локальних мережах у середині 80-х років, були звичайними, нерозподіленими прикладними програмами, бо їх було написано для автономних комп'ютерів, а потім просто перенесено в мережне середовище. Створення розподілених прикладних програм, хоч і мало багато переваг (зменшення мережного трафіка,

спеціалізація комп'ютерів), виявилося справою зовсім не простою. Потрібно було розв'язувати безліч додаткових проблем: на скільки частин розбити прикладну програму; які функції покласти на кожну частину; як організувати взаємодію цих частин, щоб у разі збоїв і відмов якихось частин, ті частини, які є працездатними, коректно завершували роботу та інших. З огляду на це дотепер тільки невелика частина прикладних програм є розподіленими, хоча очевидно, що саме за цим класом прикладних програм майбутнє, тому що вони повною мірою можуть використовувати потенційні можливості телекомунікаційних мереж з розпаралелюванням обчислень.

1.2.2. Модель клієнт-сервер

У базовій моделі **клієнт-сервер** усі процеси в розподілених системах поділяють на дві групи. Процеси, які прямо реалізують деяку службу, наприклад службу файлової системи або бази даних, є **серверами** (servers). Процеси, які вимагають служби у серверів через надсилання запиту і подальшого очікування відповіді від сервера, – **клієнти** (clients). Взаємодію між клієнтом та сервером називають режимом **запит-відповідь** (request-reply behavior).

Модель клієнт-сервер була приводом для багатьох дебатів і суперечок, у яких одне з основних питань полягало в тому, як розподілити частини програмного забезпечення між клієнтом і сервером. Зрозуміло, що зазвичай чіткого розуміння немає. Наприклад, сервер розподіленої бази даних може постійно бути клієнтом, який передає запити на різноманітні файлові сервери, відповідальні за реалізацію таблиць цієї бази даних. У такому разі сервер баз даних сам не робить нічого, крім обробки запитів.

Проте, розглядаючи велику кількість прикладних програм типу клієнт-сервер, передбачених для організації доступу користувачів до баз даних, часто рекомендують поділяти їх на три рівні:

- рівень інтерфейсу користувача;
- рівень обробки;
- рівень даних.

Рівень користувацького інтерфейсу містить усе необхідне для безпосереднього спілкування з користувачем, наприклад, для керування дисплеєм.

До рівня обробки належить прикладне програмне забезпечення, а **до рівня даних** –дані, з якими доводиться працювати.

1.3. Особливості розподілених систем

1.3.1. Переваги розподілених систем

Розрізняють низку причин, які зумовлюють заміну централізованих систем розподіленими:

- розподілені системи легко масштабуються, тобто нові вимоги до інформаційної системи у разі розширення підприємства можуть бути легко реалізованими;
- наявні рішення інтегруються, тобто такі системи можуть використовуватися системними компонентами, що приєдналися, без розробки по-новому системи з новою функціональністю;
- поступове системне розширення мінімізує ризик перевантаження окремих системних компонентів;
- організаційне керування потужністю розподіленої системи зумовлює ефективну, з погляду вартості, реалізацію, при цьому система гнучка й легко адаптується;
- власник ресурсу має можливість самостійно керувати цим компонентом та вільно втручатися й реконфігурувати систему в своїх інтересах;
- окремі складові частини розподіленої системи повністю автономні, тобто у разі помилки або навіть випадання одного з компонентів системи інші елементи можуть, не перериваючись, продовжувати роботу й, за потреби, розв’язувати проблеми, що виникають;
- розподілені системи дають можливість спільно використовувати дані та пристрої, одночасно утворюючи ілюзію безпосереднього під’єднання цих ресурсів до комп’ютера кожного користувача, тобто гнучко розподілити роботи в усій системі;
- розподілені системи забезпечують користувачам оперативний доступ до великої кількості корпоративної інформації незалежно від місця їх перебування.

Переваги розподілених апаратних рішень, які використовуються в сучасному обчислювальному обладнанні, такі:

- більш прості модулі, поєднання яких забезпечує необхідний ресурс, їх сумарна дешевизна порівняно з єдиним, більш складним, модулем;
- підвищення надійності, оскільки вихід з ладу одного модуля не призведе до необхідності ремонту чи заміни обладнання, його функції розподіляться між іншими, в результаті чого зменшиться лише продуктивність загальної схеми;
- для збільшення продуктивності загальної системи не потрібно розробляти нових апаратних елементів;
- полегшене тепловідведення – блоки «розкидано» на опорній платі;
- зменшується диференціація температурних зон і абсолютні значення температур.

1.3.2. Недоліки розподілених систем

Поряд із цими (й багатьма іншими) перевагами розподілені системи мають також і недоліки, які передусім зумовлені організацією ефективної взаємодії окремих частин розподіленої системи.

По-перше, це труднощі зумовлені програмним забезпеченням – операційними системами і прикладним програмним забезпеченням. Програмування для розподілених систем суттєво відрізняється від програмування для централізованих систем. Так, мережна операційна система, виконуючи в загальному випадку всі функції керування локальними ресурсами комп'ютера, ще й вирішує численні завдання щодо надання доступу до мережних служб. Розробка мережного прикладного програмного забезпечення ускладнюється через необхідність організувати спільну роботу його частин, які виконуються на різних машинах.

По-друге, багато проблем виникає через транспортування повідомлень каналами зв'язку між комп'ютерами, при цьому основним завданням є забезпечення надійності (щоб передані дані не губилися і не спотворювалися) та продуктивності (щоб обмін даними відбувався з прийнятними затримками). У структурі загальних витрат на обчислювальну мережу витрати на вирішен-

ня «транспортних питань» становлять велику частину, в той час як у централізованих системах таких проблем не виникає.

По-третє, проблемними є питання, зумовлені необхідністю гарантування безпеки інформації, яка циркулює в мережі, що набагато складніше вирішувати в обчислювальній мережі, ніж у централізованій системі.

Недоліки розподілених апаратних рішень, які використовують у сучасному обчислювальному обладнанні, такі:

- необхідність розробки і введення модулів синхронізації;
- необхідність розробки й реалізації алгоритмів перехоплення функцій блоків, які вийшли з ладу, ідентифікування та ізоляції некоректно працюючих блоків;
- збільшення розмірів опорних плат разом з підвищенням сумарної потужності, оскільки зростає кількість модулів.

1.4. Класифікація розподілених систем

Класифікація розподілених систем залежить від апаратного і програмного забезпечення.Розглянемо класифікацію, в основі якої лежить апаратне забезпечення. Виходячи з того, яким чином окремі компоненти розподіленої системи з'єднані між собою і як вони взаємодіють. Виходячи з цього, можливі різні підходи до класифікації.

Комп'ютери, як основні складові апаратного забезпечення, поділяють на звичайні (однопроцесорні), паралельні (багатопроцесорні) та суперкомп'ютери.

Паралельний комп'ютер – це набір процесорів, здатних спільно працювати під час розв'язання обчислювальних задач. Таке визначення достатньо широке, охоплює як паралельні суперкомп'ютери, що мають сотні або тисячі процесорів, так і мережі робочих станцій.

Суперкомп'ютер – це пристрій, який зводить проблему обчислень до проблеми введення/виведення, тобто те, що раніше довго обчислювали, іноді записуючи щось на диск, на суперкомп'ютері може виконатися миттєво, в

результаті чого найбільш неефективними стають порівняно повільні пристрої введення/виведення.

Важливою ознакою багатьох паралельних архітектур є те, що доступ до локальної пам'яті процесора дешевший, ніж доступ до віддаленої пам'яті (інших процесорів мережі). Отже, з погляду ефективності обчислень бажано, щоб доступ до локальних даних був більш частим, ніж доступ до віддалених даних. Таку властивість програмного забезпечення називають локальністю (locality), яка поряд з паралелізмом і маштабованістю є основною вимогою до паралельного програмного забезпечення.

Архітектури паралельних комп'ютерів можуть значно відрізнятися одна від одної. Паралельні комп'ютери містять три основні компоненти: процесори, модулі пам'яті, комутаційну мережу.

Комутаційна мережа з'єднує процесори один з одним та іноді також і з модулями пам'яті. Процесори, що використовуються в паралельних комп'ютерах, зазвичай такі самі, як і процесори однопроцесорних систем, хоча сучасна технологія дозволяє розмістити на мікросхемі не лише один процесор. На мікросхемі разом із процесором можуть міститися інші складові, які дають змогу підвищити ефект під час паралельних обчислень.

Однією з властивостей, за якими розрізняють паралельні комп'ютери, є кількість можливих потоків команд. Відомим прикладом є поділ на послідовні («єдиний») і паралельні («множинний») інструкції й потоки даних, що зумовило виокремлення чотирьох класів комп'ютерів:

1. Класичний von Neumann – комп'ютер позначають як SISD (Single Instruction Single Data), де використовується єдина інструкція та єдиний потік даних.

2. Паралельні й розподілені архітектури з MIMD (Multiple Instruction Multiple Data), де використовуються множинні інструкції та паралельні потоки даних.

3. Змішані форми, що належать одному з класів SIMD (єдина інструкція – паралельний потік даних) або, рідше, MISD (множинні інструкції – єдиний потік даних).

Відповідно до наведеної класифікації розрізняють такі архітектури комп'ютерів: SIMD (Single Instruction Multiple Data), MIMD (Multiple Instruction Multiple Data).

Single Instruction Multiple – комп'ютер має *N* ідентичних синхронно працюючих процесорів, *N* потоків даних і один потік команд. Кожен процесор володіє власною локальною пам'яттю. Мережа, яка з'єднує процесори, зазвичай має регулярну топологію.

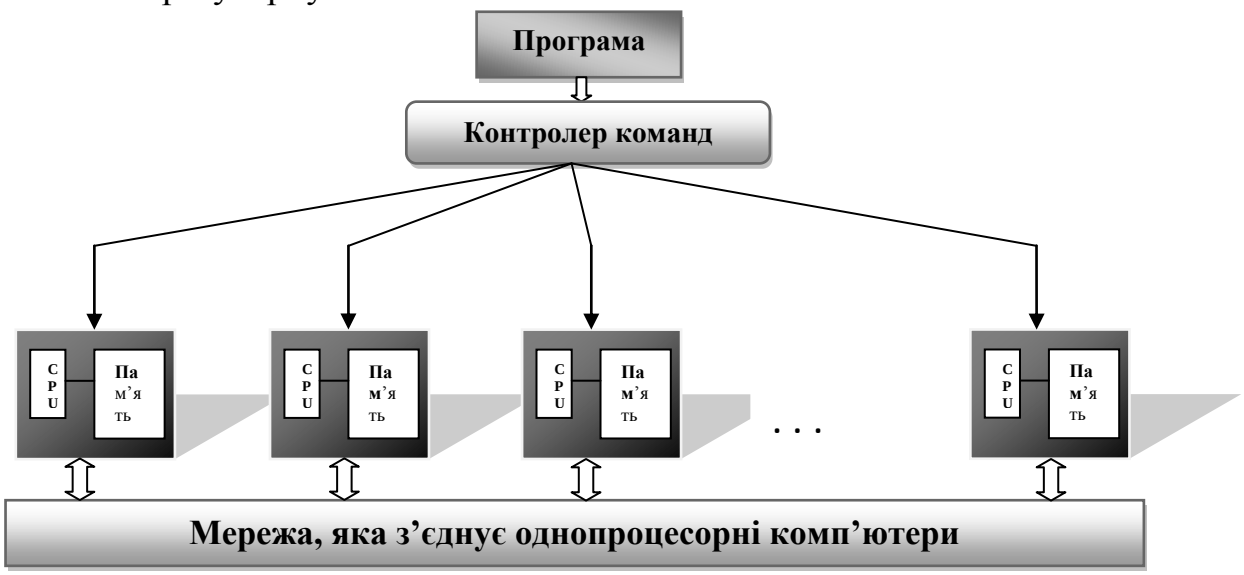


Рис. 1.2. Архітектура SIMD комп'ютера

Процесори інтерпретують адреси даних або як локальні адреси власної пам'яті, або як глобальні адреси, іноді модифіковані додаванням локальної базової адреси. Процесори отримують команди від одного центрального контролера команд і працюють синхронно, тобто на кожному кроці всі процесори виконують одну команду над даними з власної локальної пам'яті.

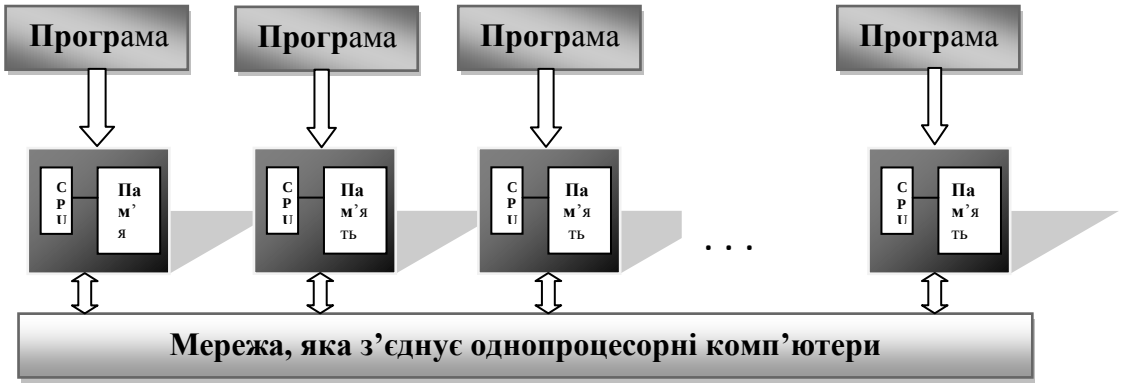


Рис. 1.3. Архітектура MIMD комп'ютера

Multiple Instruction Multiple –комп'ютер має N процесорів, N потоків команд і N потоків даних. Кожен процесор функціонує під керуванням власного потоку команд, тобто такий комп'ютер може паралельно виконувати зовсім різні програми.

Архітектури MIMD класифікують також за фізичною організацією пам'яті, способом доступу до модулів пам'яті (тобто залежно від того, чи має процесор власну локальну пам'ять, чи звертається до інших блоків пам'яті, використовуючи комутаційну мережу, чи комутаційна мережа об'єднує всі процесори із загальнодоступною пам'яттю). Виходячи з методів доступу до пам'яті, її організації, розрізняють такі типи паралельних (MIMD) архітектур:

1. Комп'ютери з розподіленою пам'яттю (distributed memory), у яких кожен процесор має доступ лише до локальної, власної пам'яті, процесори об'єднані в мережу, а доступ до віддаленої пам'яті можливий тільки за допомогою системи обміну повідомленнями.

2. Комп'ютери зі спільною пам'яттю (true shared memory) – всі процесори спільно звертаються до загальної пам'яті переважно через шину або ієрархію шин.

3. Комп'ютери з віртуальною спільною пам'яттю (virtual shared memory) – у таких системах загальної пам'яті немає, кожен процесор має власну локальну пам'ять і може звертатися до локальної пам'яті інших процесорів, використовуючи глобальну адресу. Якщо глобальна адреса вказує не на локальну пам'ять, то доступ до пам'яті реалізується за допомогою повідомлень, які пересилаються мережею, що з'єднує процесори, з малою затримкою.

Паралельні обчислювальні засоби класу MIMD утворюють три підкласи: симетричні мультипроцесори (SMP), кластери, масово паралельні системи (MPP). В основу цієї класифікації покладено структурно-функціональний підхід.

Симетричні мультипроцесори складаються із сукупності процесорів, що мають однакові можливості доступу до пам'яті й зовнішніх пристроїв і функціонують під керуванням однієї операційної системи. Частковим випадком SMP є однопроцесорні комп'ютери. Усі процесори SMP мають розділювану загальну пам'ять з єдиним адресним простором.

Кластерна система утворюється з модулів, об'єднаних системою зв'язку або розділюваними пристроями зовнішньої пам'яті, наприклад дисковими масивами. Нині для створення кластерних систем використовуються спеціалізовані фірмові засоби (наприклад, MEMORY CHANNEL фірми DEC, AWS фірми NCR), або такі універсальні локальні та глобальні мережі, як Ethernet, FDDI (Fiber Distributed Data Interface), й інші мережі, наприклад, які працюють з протоколами TCP/IP, або дискові масиви з високошвидкісними широкими подвійними (Wide/Fast) і квадро PCI SCSI контролерами. Розмір кластера варіюється від декількох до декількох десятків модулів.

Масово паралельні системи, на відміну від кластерів, мають більш швидкісні, зазвичай спеціалізовані, канали зв'язку між обчислювальними модулями, а також широкі можливості з маштабування. Крім того, у MPP фіксується деякий досить високий рівень інтерфейсу прикладних програм (API), підтримуваний розподіленою операційною системою. Однак підтримку працездатності й оптимізацію завантаження процесорів у MPP менш розвинуто порівняно з кластерами через різноманітність виконуваних програм і те, що функціональних зв'язків між програмами немає.

Приклад. Прикладом систем зі спільною пам'яттю може бути багатоядерний центральний процесор сімейства Intel Core, що має один корпус і встановлюється в один роз'єм на системній платі комп'ютера, але операційна система сприймає кожне його обчислювальне ядро як окремий процесор з повним набором обчислювальних ресурсів. Кеш-пам'ять першого рівня у кожного ядра своя, а кеш-пам'ять другого рівня поділювана – кеш розміщений на одному кристалі, так само, як обидва ядра, та доступний кожному з них у повному обсязі. На сьогодні основні виробники процесорів – Intel і AMD – визнали подальше збільшення кількості ядер процесорів одним з пріоритетних напрямів збільшення продуктивності. Вже освоєно виробництво 4, 6-ядерних процесорів для домашніх комп'ютерів, а також 8 і 10-ядерних у серверних системах.

Системи з розподіленою пам'яттю лідирують за показником пікової продуктивності, оскільки будь-які нові однопроцесорні (або багатопроцесорні на основі пам'яті) системи можуть бути легко об'єднані мережею і використані як багатопроцесорні комплекси з розподіленою пам'яттю.

До систем із розподіленою пам'яттю належать:

- трансп’ютер – паралельний обчислювальний прискорювач для комп’ютера загального призначення (HOST). Як хост-системи можуть бути робочі станції типу Sun та персональні комп’ютери IBM PC;

- гібридні системи, в основу яких покладено трансп’ютерні канали зв’язку, набули поширення, але втрачають свої позиції через недостатню пропускну здатність мережі.

- кластери робочих станцій – об’єднання певної кількості персональних комп’ютерів і/або робочих станцій.

Приклад. Прикладом систем з розподіленою пам’яттю можуть бути багатоядерні центральні процесори сімейства Athlon 64 X2 (3-10) або багатопроцесорні материнські плати для одноядерних процесорів, які обробляють задачі конвеєрно. У процесі виконання команд у конвеєрі зміщуються етапи виконання для декількох команд (тобто в конвеєрі одночасно міститься декілька команд) на різних стадіях. Кожний етап конвеєрного обчислення виконується за один машинний цикл. Зрозуміло, що виконання окремої команди на послідовному процесорі (де команда виконується зразу за один машинний цикл) може бути швидшим, аніж у конвеєрній організації, але за рахунок зміщення виконання різних команд загальна швидкодія істотно збільшується.

З погляду архітектури комп’ютера розрізняють *вільнозв’язані* й *сильнозв’язні* системи. У сильнозв’язних системах потрібно класифікувати затримку під час передачі повідомлення від одного CPU до другого як низьку, тому реалізується висока швидкість передачі. У вільноз’єднаній системі навпаки швидкість передачі низька та можливі триваліші затримки. Прикладом такої системи є два комп’ютери, з’єднані модемом і телефонною мережею. Сильнозв’язні системи оснащені зазвичай shared memory (поділюваною пам’яттю), їх називають мультипроцесорними, оскільки вони використовують паралельне розв’язання окремого завдання і можуть розглядатися як система. Обробляючи декілька незалежних завдань, необхідно використовувати мультикомп’ютерну систему з неподілюваною пам’яттю, яка і є справжньою розподіленою системою.

Кожна із цих категорій може на основі використовуваної структури мережі поділятися на шинно-базовані й switch-базовані. Під **шинно-базованою** розуміють окрему мережу, кабель або інший засіб передачі інформації, що зв’язує всі комп’ютери. **Switch-базовані** системи не мають у своєму розпоря-

дженні такої мережі, а натомість підтримують окремі зв'язки між комп'ютерами, тому в кожному вузлі мережі має міститися пристрій для здійснення процесу маршрутизації.

У мультикомп'ютерній системі значно зменшується вихід повідомлень, тому що локальна пам'ять не так завантажена. Для зменшення кількості затримок усі процесори не зв'язуються Switch безпосередньо, але можуть лише опосередковано обмінюватися повідомленнями з іншим процесором. Приклади розглянутих топологій наведено на рис. 1.4.

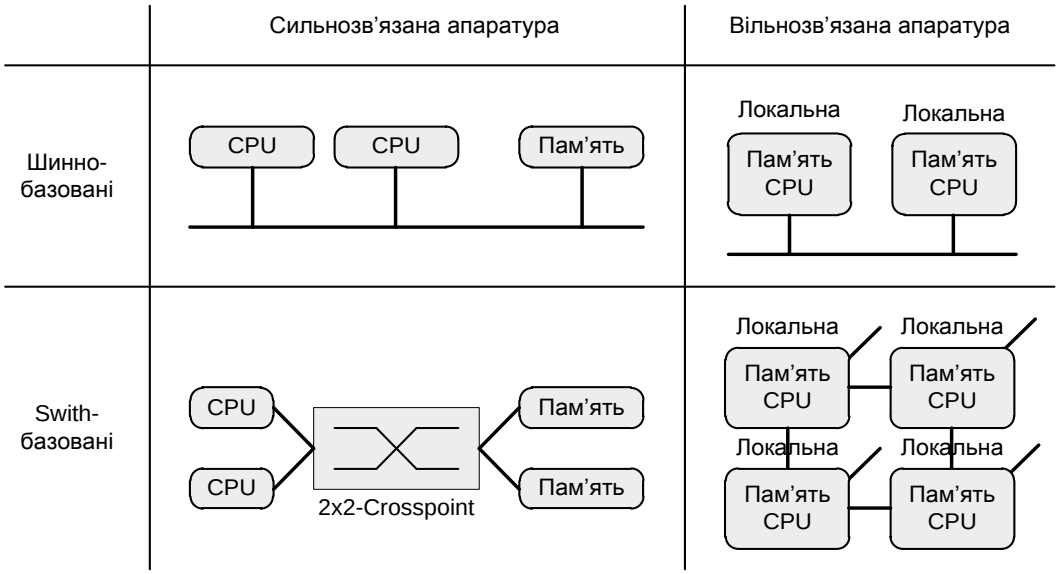


Рис. 1.4. Шинно- і switch-базовані системи зі спільною пам'яттю і без неї

Таким чином, класифікацію здійснено винятково за ознаками реалізації апаратної частини; системи поділено на чотири групи: шинно- й відповідно Switch-базовані, вільноз'єднані або сильнозв'язні.

Аналогічно до класифікації за апаратною складовою, використовують класифікацію, в основу якої покладено характеристики програмного забезпечення, за ступенем його зв'язності. Залежно від їх архітектури системи поділяють на **вільнозв'язані** та **сильнозв'язні**.

Вільнозв'язане програмне забезпечення дозволяє комп'ютерам і користувачам розподіленої системи насправді виконувати незалежну одну від іншої роботу й тільки в деяких випадках, якщо буде потреба, сумісно використовувати обладнання.

Приклад: оператори персонального комп'ютера із власним CPU, власною пам'яттю й операційною системою, які спільно використовують лазерний принтер.

Сильнозв'язне програмне забезпечення реалізує одну програму на різних комп'ютерах одночасно.

Для практичної реалізації розподілених систем було створено спочатку **мережну операційну систему**, потім **мультипроцесорну операційну систему** й **розподілену операційну систему**.

Мережна операційна система передбачає, що кожний користувач має робочу станцію (Workstation) із власною операційною системою, в цьому разі комунікація використовується для доступу до спільних файлів.

Мультипроцесорну операційну систему використовують часто для спеціальних цілей, зокрема для системи ведення банку даних. Характерною ознакою цієї системи є наявність окремого процесу в спільній пам'яті. Комунікація між окремими компонентами такої системи відбувається для узгодження процесів інформаційного обміну.

Розподілена операційна система створює для користувача ілюзію, що вся мережа – це єдиний великий комп'ютер, де зберігаються вся інформація й усі прикладні програми. Комунікація в такій системі необхідна для обміну повідомленнями.

Розподілені системи поділяють на такі категорії:

Периферійні системи, які являють собою групу машин, що мають спільне функціональне призначеня і зв'язані з однією (зазвичай більш великою) машиною. Периферійні обчислювальні процеси поділяють своє навантаження із центральним процесором і переадресовують йому всі звертання до операційної системи. Мета периферійної системи полягає у підвищенні загальної продуктивності мережі й у наданні можливості виділення процесора для одного процесу в операційному середовищі. Система запускається як окремий модуль і на відміну від інших моделей розподілених систем, не має реальної автономії, за винятком процесів диспетчиризації та розподілення локальної пам'яті.

Розподілені системи типу «Newcastle», які дозволяють налагоджувати дистанційний зв'язок за іменами дистанційних файлів у бібліотеці. Дистанційні файли мають специфікацію (складне ім'я), яка у вказівнику шляху

пошуку містить спеціальні символи або додатковий компонент імені, яка передує кореню файлової системи. Реалізують цей метод без внесення змін у ядро системи, внаслідок чого він більш простий, ніж інші методи, але менш гнучкий.

Абсолютно прозорі розподілені системи, в яких для звертання до файлів, розташованих на інших машинах, достатньо вказати стандартні складні імена файлів, розпізнання цих файлів як дистанційних є функцією ядра. Маршрути пошуку файлів вказано в їх складних іменах.

1.5. Характеристики розподілених систем

Розподілені системи мають такі характерні риси:

– просторова розподіленість компонент розподіленої системи, тобто вони взаємодіють або локально, або віддалено;

– компоненти розподіленої системи можуть працювати паралельно, через що швидкість роботи зростає порівняно з послідовною роботою;

– кожний стан компонента розглядається локально, тобто з погляду певного обчислювального процесу, запущеного з локального робочого місця.

– компоненти працюють окремо й можуть «випадати», не руйнуючи системи в цілому, незалежно одна від одної, таким чином розподілені системи підлягають частковому системному «випаданню»;

– система працює асинхронно, процеси комунікації й обробки не керуються глобальним системним часом, змінні й процеси синхронізуються;

– у розподіленій системі функції керування розподіляються між різними автономними компонентами, оскільки жодна окрема компонента не може здійснювати весь контроль, що гарантує певний рівень автономії;

– розподілена система може утворюватися як об’єднання наявних систем, тобто, потрібно контекстно-повне керування іменами, що дає можливість однозначно інтерпретувати найменування в рамках адміністративної або технологічної області –такий випадок вважають федеративним керуванням іменами;

– для підвищення потужності розподіленої системи, програми й дані можна переміщувати між різними вузлами, таку концепцію називають **міграці-**

єю, для підтримки якої використовують додаткові механізми, які протоколюють стан та розташування програм і даних;

– розподілена система має використовувати динамічні зміни структури, ця **динамічна реконфігурація** потрібна, наприклад, тоді, коли протягом певного часу очікуються нові з'єднання;

– архітектура комп'ютерів може використовувати різні топології й механізми, зокрема якщо апаратура надходить від різних виробників, то цю особливість називають гетерогенністю;

– розподіленій системі властива еволюція, вона може змінюватися протягом часу її життя;

– джерела відомостей, одиниці обробки й користувачі можуть бути фізично мобільні, а програми й дані можуть переміщуватися між вузлами для одержання даних та ресурсів з інших вузлів системи або підвищення потужності.

Для досягнення цих характерних рис слід виконувати певні вимоги, які висуваються до розподілених систем: прозорість, відкритість, гнучкість, масштабованість, стійкість, безпека, ефективність.

Прозорість є основною вимогою, яка спрощує взаємодію розподілених прикладних програм і приховує фізичне розподіленння процесів та ресурсів серед багатьох комп'ютерів. Цей принцип дозволяє замовчувати складність реалізації розподілених систем від користувача та внутрішні процеси в системі за допомогою прозорих для нього функцій, унаслідок чого розподілена система для користувача стає більш зручною. Розподілені системи, які уявляються користувачам і прикладним програмам у вигляді єдиної комп'ютерної системи, називають прозорими (transparent).

Розрізняють багато характерних рис прозорості розподілу, серед яких відокремимо такі:

– прозорість доступу приховує спеціальні механізми доступу для локальних або віддалених служб до ресурсів системи;

– прозорість місця розташування приховує від користувача системні топології, тобто приховує фізичне місце розташування ресурсів (досягається

присвоєнням логічних імен, дозволяє замовчувати різні формати даних, які передаються різним процесам);

– прозорість міграції приховує факт переміщення ресурсів у інше місце;

– прозорість виконання приховує спосіб обробки обчислювальних потоків, коли обчислення виконуються паралельно або послідовно;

– прозорість зміни місця розташування приховує зміну ресурсу під час його обробки;

– прозорість реплікації приховує факт реплікації під час обробки даних;

– прозорість паралельного доступу приховує факт спільного використання ресурсів декількома конкуруючими процесами користувачів;

– прозорість відмови приховує факт виникнення відмов у системі й відновлення ресурсу після їх локалізації;

– збереження приховує факт розташування ресурсу на носіях або кешування ресурсу;

– прозорість з'єднання приховує межі між адміністративними й технологічними областями в системі;

– прозорість групи приховує користувачів від групи під час колективної роботи із системою.

Відкрита розподілена система (open distributed system) – це система, що пропонує стандартні засоби та служби доступу до системи широкому колу користувачів, які використовують стандартні синтаксис і семантику всіх протоколів взаємодії. Всі протоколи взаємодії компонентів усередині розподіленої системи в ідеальному випадку ґрунтуються на загальнодоступних стандартах, що дозволяє використовувати для створення компонент різні засоби розробки й різні операційні системи, а кожна компонента має точну й повну специфікацію своїх сервісів. За таких умов компоненти розподіленої системи можуть бути створені незалежними розробниками. У разі порушення цієї вимоги може стати неможливим поява розподіленої системи, яка охоплює кілька незалежних організацій.

Важливою характеристикою, що забезпечує відкритість розподілених систем, є наявність загальних специфікацій інтерфейсів, які підтримуються

службами розподілених систем для реалізації інтерфейсів різними виробниками програмного забезпечення. Такі специфікації інтерфейсів є однозначними для різних реалізацій конкретного інтерфейсу.

Гнучкість – простота конфігурування системи, яка складається з різних компонентів різних виробників; можливість перенесення системи з одних операційних систем на інші, навіть в іншу файлову систему. Не має виникати ускладнень під час введення в систему нових компонентів або заміни наявних, при цьому інші компоненти, з якими не проводилося жодних дій, залишаються незмінними. Для забезпечення масштабованості слід використовувати децентралізовані дані, алгоритми, служби. У разі географічної масштабованості збільшується ризик атакування ресурсів з боку територіально віддалених користувачів, а процедура узгодження вимагає реєстрації користувачів у одному інформаційному просторі власної області або домену, а також реєстрації користувачів одного домену в іншому з наданням їм відповідних прав доступу. Для забезпечення гнучкості розподіленої системи її слід будувати за принципом модульності, коли кожний її компонент – це певний модуль, що має своє функціональне призначення. У той же час кожний компонент підтримує принцип федеративності для співіснування автономних модулів на підставі певних стандартизованих правил. Така побудова розподілених систем забезпечує ефективну керованість, надійність функціонування системи й підтримку значень показників функціонування на заданому рівні, зокрема швидкодії.

Масштабованість обчислювальних систем має кілька аспектів, найбільш важливим з яких є можливість долучити до розподіленої системи нові комп’ютери для підвищення продуктивності системи, зумовленого необхідністю балансування навантаження (load balancing) на сервери системи. Проблема масштабування має враховувати ефективність розподілу ресурсів серверів, які обслуговують запити клієнтів.

Масштабованість системи може вимірюватися за трьома різними показниками: по-перше, система може бути масштабованою за розміром, що означає легкість під’єднання до неї додаткових користувачів і ресурсів; по-друге, система може масштабуватися географічно, тобто користувачі й ресурси

можуть бути рознесені у просторі; по-третє, система може бути масштабованою адміністративно, тобто бути простою в керуванні під час її роботи в адміністративно незалежних організаціях. На жаль, система, якій властива масштабованість за одном або декількома із цих параметрів, у разі масштабування часто втрачає продуктивність.

Основні підходи, що забезпечують масштабованість розподіленої системи:

1. **Приховування від користувача часу очікування зв'язку** передбачає можливість не враховувати час очікування відповіді від віддаленого сервера, але прикладні програми розраховують на синхронний вид зв'язку, коли передбачається переривання активного процесу під час відповіді на раніше надісланий запит. За таких умов прикладна програма викликає оброблювача запитів, який і погоджує всі обчислювальні процеси в системі й забезпечує логічну цілісність даних. За рахунок такого принципу обробки запит користувача в розподіленій системі має бути або коректно виконаний повністю, або не виконаний взагалі. Ситуація, коли частина компонент системи коректно обробила запит, що надійшов, а частина – ні, є найгіршою й неможливою. Такий підхід характерний для розподілених систем, які використовують технологію обробки пакетів у процесі передачі інформації з мережі.

2. Розподіл передбачає **розбиття компонентів системи на дрібні частини** й подальше рознесення цих частин фізичними територіально розподіленими вузлами системи. Прикладом розподілу є система доменних імен Internet (DNS).

3. **Реплікація** не тільки підвищує доступність мережних ресурсів, але й допомагає вирівняти завантаження компонентів системи, що приводить до підвищення продуктивності. **Кешування** (caching) є особливою формою реплікації, причому розбіжності між ними нерідко малопомітні або взагалі штучні, оскільки і під час реплікації, результатом кешування є створення копії ресурсу зазвичай у безпосередній близькості від клієнта, який використовує цей ресурс. Кешування – це дія, яку виконує споживач ресурсу, а не його власник. На масштабованість може погано вплинути один істотний недолік кешування й реплікації, оскільки створюється низка копій ресурсу, модифікація однієї копії робить її відмінною від інших, відповідно, кешування й реплікація зумовлюють проблеми несуперечності даних (consistency).

Стійкість. Під стійкістю розуміємо можливість дублювання декількома комп'ютерами функцій або ж можливість автоматичного розподілу функцій усередині системи у разі виходу з ладу одного з комп'ютерів. У ідеальному випадку немає унікальної точки збою, тобто вихід з ладу будь-якого комп'ютера не заважає обслуговуванню запитів користувача. Стійкість розподіленої системи пов'язана з поняттям масштабованості, але не еквівалентна йому.

Приклад. Припустімо, система використовує набір серверів, які обробляють запити, й одного диспетчера запитів, який розподіляє запити користувачів між серверами. Таку систему можна вважати добре масштабованою, однак диспетчер є вразливим місцем такої системи. Натомість, система з єдиним сервером може бути стійкою, якщо наявний механізм його автоматичної заміни у разі виходу з ладу.

Виконання вимог стійкості й масштабованості зазвичай спричиняє деякі додаткові витрати, що на практиці не завжди доцільно. Однак технології, які використовують під час побудови розподілених систем, мають допускати принципову можливість створення стійких і високомасштабованих систем.

Безпека. Для кожного компонента, який утворює розподілену систему, має бути використана функція авторизації не тільки для компонент, які його викликають, але і для користувачів. Дані, передані між компонентами, слід захистити як від спотворення, так і від перегляду третіми сторонами.

Ефективність. Під ефективністю розподілених систем будемо розуміти мінімізацію накладних витрат, спричинених розподіленим характером системи. Оскільки ефективність у такому вузькому значенні може суперечити безпеці, відкритості й надійності системи, то вимога ефективності в цьому контексті є найменш пріоритетною. Наприклад, на підтримку логічної цілісності даних у розподіленій системі можуть витрачатися значні ресурси часу й пам'яті, однак система з недостовірними даними не потрібна користувачам, тому такі додаткові витрати є виправданими.

1.6. Висновки

Розподілені системи складаються з автономних комп'ютерів, які працюють спільно, створюючи уявлення про роботу в єдиній зв'язній системі. Їх перевага полягає в тому, що вони спрощують інтеграцію різних приклад-

них програм, які працюють на різних комп'ютерах, у єдину систему, добре масштабуються за умови правильного проектування.

Розмір розподілених систем обмежується тільки розміром базової мережі.

Розподілені системи класифікують за принципами, в основі яких лежать характеристики апаратного та програмного забезпечення.

Реалізація розподілених систем характеризується складністю програмного забезпечення, зниженням продуктивності й наявністю проблем з безпекою.

Розподілені операційні системи використовують для керування апаратним забезпеченням взаємозалежних комп'ютерних систем та сприймають у вигляді єдиної системи.

Мережні операційні системи ефективно поєднують різні комп'ютери, які працюють під керуванням своїх операційних систем, забезпечуючи користувачам доступ до локальних служб кожного з вузлів. Однак мережні операційні системи не створюють у користувача відчуття роботи з єдиною системою, що характерно для розподілених операційних систем.

1.7. Запитання для самоконтролю

1. Дайте визначення поняттю «розподілена система».
2. Яким є основне завдання розподіленої системи?
3. Коли та чим була зумовлена відмова від централізації?
4. Назвіть переваги розподілених систем.
5. Які недоліки мають розподілені системи?
6. Наведіть класифікацію розподілених систем відповідно до апаратного забезпечення.
7. Наведіть класифікацію розподілених систем відповідно до програмного забезпечення.
8. Які характерні риси мають розподілені системи?
9. Яким вимогам мають задовольняти розподілені системи? Коротко охарактеризуйте кожну з них.
10. Дайте визначення поняттю «суперкомп'ютер».

11. Назвіть основні компоненти паралельних комп’ютерів.

12. Чому апаратура комп’ютерів розвивається у напрямку «розподілення»?

13. Наведіть приклади реалізації апаратури зі спільною памяттю для ПК.

14. Наведіть приклади реалізації апаратури з розділюваною пам’яттю для ПК.

2. РОЗПОДІЛЕНЕ СЕРЕДОВИЩЕ

2.1. Концепції апаратних рішень

Незважаючи на те, що всі розподілені системи містять по декілька процесорів, є різні способи їх об'єднання в систему. Особливо це стосується варіантів їх з'єднання й організації взаємного обміну.

Розглянемо коротко апаратне забезпечення розподілених систем, зокрема варіанти з'єднання машин між собою. За минулі роки було запропоновано різні класифікації комп'ютерних систем з декількома процесорами, але жодна з них не стала дійсно популярною й поширеною.

Розглянемо системи, до яких входять незалежні комп'ютери, які можна поділити на дві групи. Системи, у яких комп'ютери використовують *пам'ять спільно*, називають **мультипроцесорами (multiprocessors)**, а ті, що *працюють кожен зі своєю пам'яттю* – **мультикомп'ютерами (multicomputers)**. Основною відмінністю є те, що мультипроцесори мають *єдиний адресний простір*, який спільно використовують усі процесори. У мультикомп'ютерах кожна машина використовує свою власну пам'ять.

Кожну із цих категорій поділяють на додаткові підкатегорії за архітектурою мережі, яка їх з'єднує. Найчастіше використовують дві архітектури – *шинну (bus)* і *комутовану (switched)*. Під **шинною** розуміють *плату, шину, кабель або інше середовище*, що з'єднує всі машини між собою. **Комутовані системи,** на відміну від шинних, не мають єдиної магістралі, такої як, наприклад, у кабельного телебачення. Замість неї від машини до машини тягнуться окремі канали, виконані із застосуванням різних технологій зв'язку. Повідомлення передаються каналами з узгодженням рішення про комутацію з конкретним вихідним каналом для кожного з них.

Також розподілені комп'ютерні системи поділяють на *гомогенні (homogeneous)* і *гетерогенні (heterogeneous)*. Цей поділ стосується винятково мультикомп'ютерних систем. Для гомогенних мультикомп'ютерних систем характерна одна мережа, яка з'єднує комп'ютери та використовує єдину технологію. Однакові також усі процесори, які зазвичай мають доступ до одна-

кових обсягів власної пам’яті. Гомогенні мультикомп’ютерні системи нерідко використовують як паралельні (працюють з одним завданням), за аналогією до мультипроцесорних.

На відміну від гомогенних, гетерогенні мультикомп’ютерні системи можуть містити декілька незалежних комп’ютерів, з’єднаних різноманітними мережами.

Приклад. Розподілена комп’ютерна система може бути побудована з декількох локальних комп’ютерних мереж, з’єднаних комутованою магістраллю FDDI або ATM.

Мультипроцесорні системи мають одну характерну рису: всі процесори мають прямий доступ до загальної пам’яті. Мультипроцесорні системи шинної архітектури складаються з деякої кількості процесорів, приєднаних до загальної шини, а через неї – до модулів пам’яті. Найпростіша конфігурація містить плату із шиною або материнською платою, у яку вставляють процесори й модулі пам’яті. Проблема такої схеми полягає в тому, що у разі наявності 4-х або 5-ти процесорів шина стає перевантаженою і продуктивність різко знижується. Щоб розв’язати цю проблему, потрібно розмістити між процесором і шиною високошвидкісну кеш-пам’ять (cache memory).

На відміну від мультипроцесорів, побудувати гомогенну мультикомп’ютерну систему порівняно нескладно. Кожний процесор прямо зв’язаний зі своєю локальною пам’яттю, а єдиною проблемою є взаємодія процесорів між собою, за таких умов необхідна схема з’єднання, але оскільки у такому разі ключовим моментом є тільки наявність зв’язку між процесорами, то обсяг трафіку буде на кілька порядків нижчим, ніж у випадку використання мережі для підтримки трафіку між процесорами й пам’яттю.

Найбільшу кількість наявних нині розподілених систем побудовано за схемою гетерогенних мультикомп’ютерних систем. Це означає, що комп’ютери, які є частинами цієї системи, можуть відрізнятися, наприклад за типом процесора, розміром пам’яті й продуктивністю каналів введення-виведення. На практиці роль деяких із цих комп’ютерів можуть виконувати високопродуктивні паралельні системи, зокрема мультипроцесорні або гомогенні мультикомп’ютерні. Мережа, яка їх поєднує, може бути дуже неоднорідною. При-

кладом гетерогенності є створення великих мультикомп'ютерних систем, які використовують наявні мережі й канали.

Приклад. Нині є фактом існування кампусних університетських розподілених систем, що складаються з локальних мереж різних факультетів, з'єднаних між собою високошвидкісними каналами. У глобальних системах різні станції можуть, у свою чергу, з'єднуватися загальнодоступними мережами, наприклад мережними службами, пропонованими комерційними операторами зв'язку, зокрема ATM або Frame relay.

2.2. Архітектура багатопроцесорних систем

Серед усієї різноманітності багатопроцесорних систем розглянемо системи типу MIMD (множинні потоки команд і множинні потоки даних). Обчислювальні системи типу MIMD являють собою об'єднання процесорів, на кожному з яких може бути запущена своя, відмінна від інших, програма, що оперує власними даними. З огляду на це MIMD системи можна поділити на два класи: *зі спільною пам'яттю і з роздільною пам'яттю.*

2.2.1. Системи зі спільною пам'яттю

У системах зі спільною пам'яттю програма поділена на процеси, які взаємодіють і автоматично розподіляються між доступними процесорами системи. Для систем цього класу є засоби автоматизованої побудови паралельних програм – векторизуючі компілятори. Достатньо широке коло послідовних алгоритмів може бути успішно адаптовано до функціонування на системах зі спільною або з розподільною пам'яттю (рис. 2.1). Система зі спільною пам'яттю зручна тому, що функціонує під єдиною копією операційної системи й не вимагає індивідуального налаштування кожного процесорного вузла. Такі комп'ютери, керовані Unix або подібними операційними системами, обслуговують користувачів у режимі колективного користування, автоматично розподіляючи потік завдань прикладного програмного забезпечення процесорами без участі адміністратора або оператора.

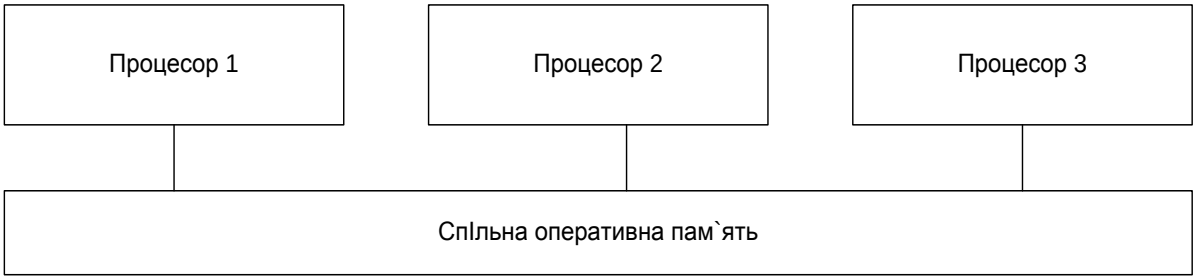


Рис. 2.1. Багатопроцесорна система зі спільною пам’яттю

Однак системам зі спільною пам’яттю властива низка істотних недоліків:

– порівняно невелика кількість процесорів;

– немає можливості збільшення кількості процесорів, тобто масштабувати систему;

– пікова продуктивність найбільших систем зі спільною пам’яттю нижча від пікової продуктивності найбільших систем з роздільною пам’яттю;

– висока вартість порівняно з аналогічними за продуктивністю системами з роздільною пам’яттю.

Ефективно об’єднати більше 20–30 процесорів на основі загальної пам’яті важко, оскільки кожний процесор повинен мати фізичний доступ до кожного з блоків оперативної пам’яті.

Приклад. У разі використання роздільних 32-розрядної адресної шини й 64-розрядної шини даних потрібною є як мінімум 96-розрядна високошвидкісна лінія доступу до пам’яті від кожного процесора, що становить технологічну проблему.

Необхідність забезпечення обміну зі швидкістю 500 Мбайт/с і вище обмежується фізичною відстанню від кожного процесора до кожного блока пам’яті. Зазвичай системи зі спільною пам’яттю виглядають досить компактно, розміщуючись в одному корпусі. Цей фактор є обмеженням під час збільшення загальної кількості процесорів через кількість блоків пам’яті, які можна розмістити в межах, заданих максимальною відстанню від блока пам’яті до процесора. Спільна пам’ять передбачає можливість усіх процесорів одночасно прочитати або записати різні дані з одного блока пам’яті, для цього кожний блок пам’яті повинен мати таку кількість точок входу, яка дорівнює кількості процесорів.

Як приклад можливого розв’язку цієї проблеми наведемо схему мережі типу «метелик» (рис. 2.2). Згідно із цією схемою вісім процесорних модулів

можуть отримати доступ до восьми блоків пам’яті через систему комутаторів, однак у кожен конкретний момент до кожного блока пам’яті може мати доступ тільки один із процесорів, що в цілому знижує продуктивність системи.

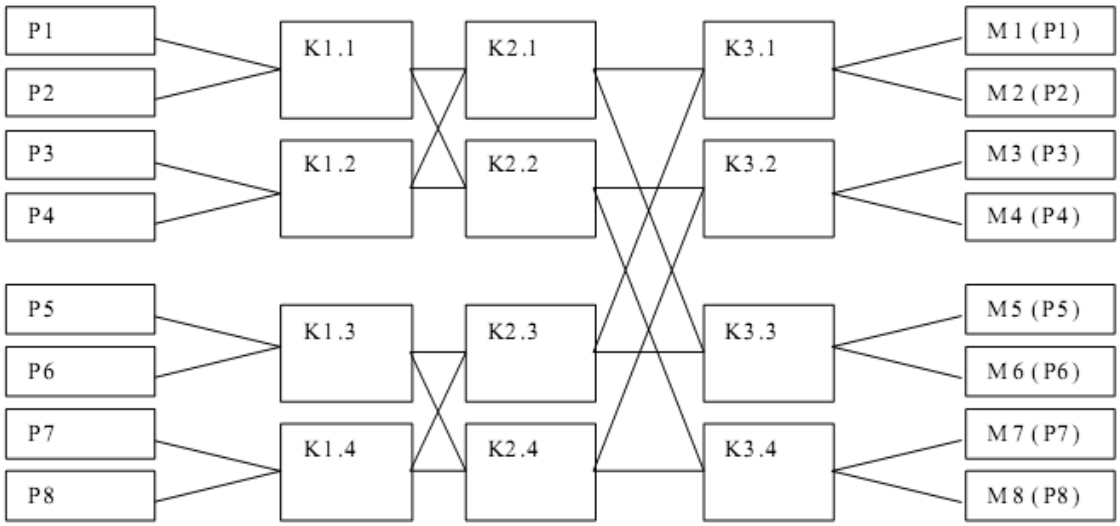


Рис. 2.2. Комунікаційна мережа типу «метелик»

Деякі сучасні системи побудовано за кластерним принципом, за яким процесори й оперативну пам’ять поділяють на кілька груп – **кластерів**. Усередині кластера процесори мають швидкий доступ до оперативної пам’яті. Доступ процесорів одного кластера до оперативної пам’яті, розташованої в іншому кластері, так само можливий, але час доступу при цьому значно зростає.

Конфлікти, що неминуче виникають під час запису різними процесорами тих самих даних, мають фундаментальний характер і знижують продуктивність системи за рахунок утрат на синхронізацію, незалежно від конкретної апаратної реалізації системи. У такий спосіб сконструйована система зазвичай не передбачає можливості суттєвого збільшення кількості процесорних вузлів і зумовлює високу вартість.

2.2.2. Системи з роздільною пам’яттю

Масштабовані системи масового паралелізму конструюють на основі об’єднання каналами передачі даних процесорних вузлів, які мають свою локальну оперативну пам’ять, недоступну іншому процесору (рис. 2.3).

Рис. 2.3. Багатопроцесорна система з роздільною пам’яттю

Обмін даними між процесорами у разі такого підходу можливий лише за допомогою повідомлень, переданих каналами зв’язку. Така схема має низку переваг порівняно із системами, побудованими на основі загальної спільної пам’яті. Зазначимо основні з них:

– порівняно низька вартість – найменший показник серед показників «ціна/продуктивність»;

– масштабованість – можливість побудови систем необхідної продуктивності, зазвичай підвищення їх потужності за рахунок установки додаткових процесорів.

Системи з роздільною пам’яттю лідирують за показником пікової продуктивності, оскільки будь-які нові однопроцесорні (або багатопроцесорні на основі пам’яті) системи можуть бути легко об’єднані мережею й використані як багатопроцесорні комплекси з роздільною пам’яттю. Проте, на жаль, ефективне використання систем з розподіленою пам’яттю вимагає значних зусиль з боку розробників прикладного забезпечення й можливе далеко не для всіх типів завдань, оскільки не для всіх послідовних алгоритмів, що добре зарекомендували себе, вдається побудувати ефективні паралельні аналоги.

Трансп’ютери. Саме від трансп’ютерів почалося масове поширення багатопроцесорних систем. Типова трансп’ютерна система є фактично паралельним обчислювальним прискорювачем для певного комп’ютера загального призначення – хостом комп’ютера (HOST). Функції хост-систем виконують також робочі станції типу Sun, персональні комп’ютери IBM PC (рис. 2.4).

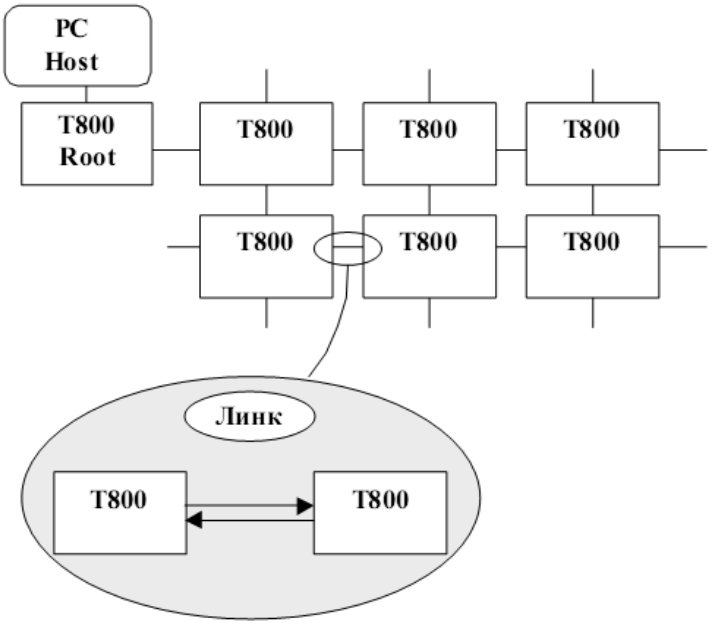


Рис. 2.4. Трансп’ютерна система й канали зв’язку

Кожний трансп’ютер виробництва фірми INMOS (Великобританія) є повнофункціональним процесором (рис. 2.5), який відрізняється від звичайного процесора тим, що містить чотири канали міжпроцессорного обміну даними – лінка (*link*).

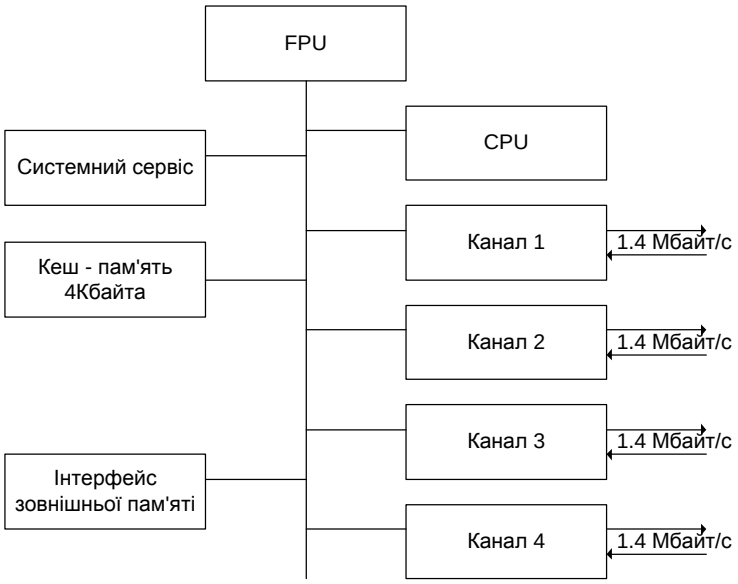


Рис. 2.5. Структура трансп’ютера типу Т-800

Кожний *link* – це обладнання синхронного небуферизованого послідовного обміну даними 4-провідною лінією зв’язку (порівняйте з 96-провідною

лінією доступу до оперативної пам’яті в системах зі спільною пам’яттю). Саме завдяки можливості з’єднувати між собою довільну кількість процесорів, керованих однією або декількома хост-системами, трансп’ютерні системи набули значного поширення.

Трансп’ютер типу T800 містить обладнання обробки цілих чисел (CPU), дійсних чисел і чотири двонаправлені канали передачі даних – лінка. Кожна з названих шістьох апаратних засобів може працювати одночасно з іншими й незалежно від них. Виконуючи запити на обробку або передачу даних, кожне з них звертається до змінних, розташованих у доступній усім спільній оперативній пам’яті, при цьому цілком можлива ситуація, коли кілька апаратних засобів одночасно звертаються до однієї й тієї ж змінної. Запити будуть оброблені коректно, коли дані тільки зчитуються, але якщо хоча б один з апаратних засобів записує дані, то результат виконання операцій читання/запису стає нез’ясованим. Якщо не визначено результату одночасного запису кількома апаратними засобами різних значень у ту саму змінну, то кожний трансп’ютер є системою зі спільною пам’яттю.

У перших трансп’ютерних системах трансп’ютери з’єднувалися лінками безпосередньо між собою й утворена в такий спосіб конфігурація, або **топологія**, фіксувалася на час розв’язування задачі. У разі потреби в отриманні іншої топології *link* доводилося з’єднувати вручну. Щоб спростити обслуговування й забезпечити можливість зміни топології системи безпосередньо у процесі обчислень було розроблено електронний комутатор C-004 (рис. 2.6), як електронно-конфігурований комутатор, що дозволяє задавати довільні парні зв’язки між 32 входами трансп’ютерних *link* і 32 виходами. На рис. 2.6 вісім трансп’ютерів жорстко з’єднано в лінійку (pipe) двома із чотирьох своїх *link*, а іншими під’єднано до комутатора. Між під’єднаними до комутатора *link* можна програмно визначити будь-які чотири зв’язки. Наприклад, показаний пунктиром зв’язок перетворює вихідну топологію «лінійка» в топологію «кільце».

Маючи порівняно невелику продуктивність, транп’ютери швидко втратили своє значення як обчислювальні елементи, але довгий час зберігали його як комутаційні у складі гібридних обчислювальних систем.

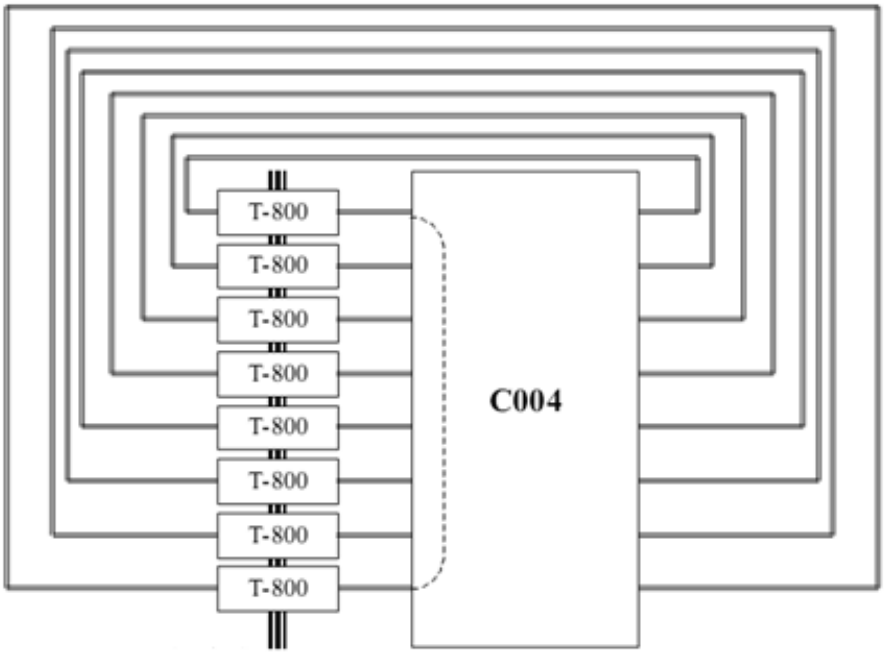


Рис. 2.6. Електронно-реконфігуроване з'єднання трансп'ютерів за допомогою комутатора C004

Гібридні системи. Розглянемо багатопроцесорну систему Parsytec PowerXplorer, яка складається з 12 процесорних вузлів (рис. 2.7), кожний з них використовує процесор Powerpc-601. Реальна продуктивність кожного процесора – близько 30 Mflops. Кожний процесор використовує 8 Мбайт локальної оперативної пам'яті, до якої має доступ трансп'ютер типу T805. Трансп'ютери за допомогою *link* з'єднані між собою топологією у вигляді прямокутної сітки, частина трансп'ютерів має вихід через один зі своїх каналів зв'язку на керувальну машину типу Sunsparc (рис. 2.8). Підключення здійснюється за допомогою SCSI-пристрою Scsi-Bridge, який підтримує чотири трансп'ютерних *link* для зв'язку із системою PowerXplorer. Реальна продуктивність кожного з чотирьох трансп'ютерных каналів зв'язку – від 1 Мбайт/с у разі передачі довгих повідомлень до 0,3 Мбайт/с у разі передачі повідомлень довжиною в декілька байтів.

Розробка програмного забезпечення для подібних систем можлива засобами системи програмування PARIX™, яка дозволяє використовувати будь-яку зручну віртуальну топологію (тор, решітку, зірку тощо), при цьому трансп'ютер прозорий для програміста. Фізична топологія жорстко задана й не змінюється.

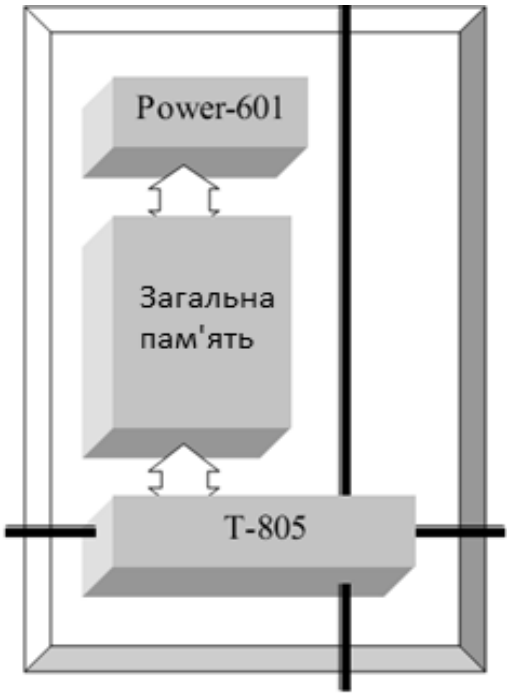


Рис. 2.7. Структура вузла PowerXplore

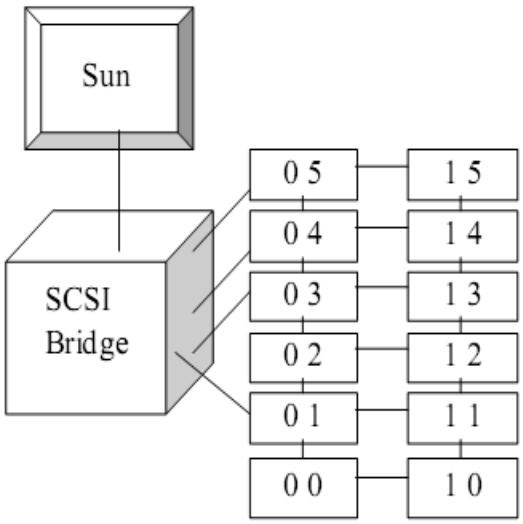


Рис. 2.8. Підключення 12-процесорної системи PowerXplorer
до керувальної машини типу Sun

Гібридні системи, в основі яких лежать трансп’ютерні канали зв’язку, набули поширення, але нині вони втрачають своє значення через недостатню пропускну здатність *link.*

Кластери робочих станцій. З появою та розвитком локальних мереж з’явилися кластери робочих станцій, які являють собою об’єднання невеликої

кількості персональних комп'ютерів і/або робочих станцій. Хоч кластери робочих станцій є відносно дешевими, ці системи часто мають нижчу ефективність розв'язання прикладних задач порівняно зі спеціалізованими системами з таких основних причин:

– певні поширені локальні мережі (Ethernet, Token Ring) не підтримують одночасної передачі даних між різними парами комп'ютерів у межах одного сегмента мережі. Це означає, що дані між комп'ютерами *c* і *d* можуть бути передані тільки після передачі даних між комп'ютерами *a* і *b*, що зменшує не дуже високу наявну швидкість передачі даних у таких мережах;

– майже завжди на робочих станціях, поєднаних у кластер, продовжують виконуватися послідовні завдання користувачів, у результаті чого менш завантажені процесори змушені очікувати більш завантажених, що призводить до загального зниження продуктивності кластера під час розв'язування паралельної задачі до рівня, визначеного найбільш завантаженою машиною;

– наявність у мережі файлових серверів призводить до нерегулярної зміни обсягів даних, які передаються через локальну мережу, що може суттєво збільшувати час обміну повідомленнями між процесорами та інтервали їх простоювання.

Системи на основі високошвидкісних мереж. Найбільш перспективними вважаються багатопроцесорні системи, побудовані на основі спеціалізованих високошвидкісних мереж передачі даних, серед яких є подані на рис. 2.9, 2.10 системи Parsytec CC (Cognitive Compute – комп'ютер розумний).

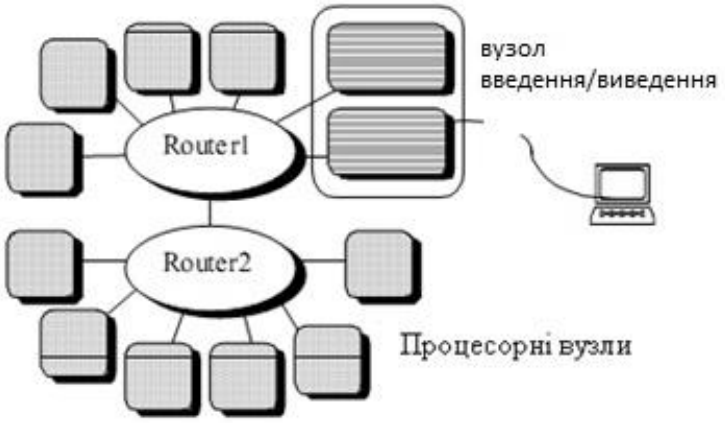


Рис. 2.9. Структура системи Parsytec CC-12

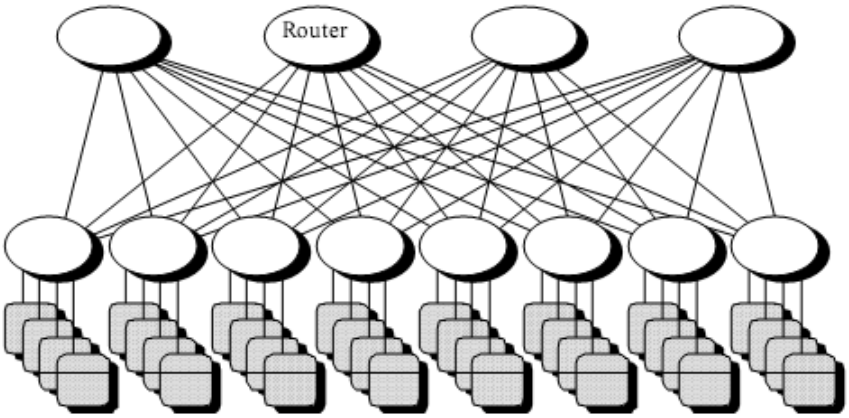


Рис. 2.10. Структура системи Parsytec CC-32

Кожний вузол систем цієї серії (рис. 2.11) являє собою повноцінний комп'ютер, керований Unix-подібною операційною системою AIX, система має вузли двох типів: обчислювальні й вузли введення/виведення. Процесорний вузол містить обчислювальний процесор Power PC-604, накопичувач на жорсткому диску, адаптер локальної мережі Ethernet, адаптер високошвидкісної мережі HS-Link. До складу вузла введення/виведення додатково входять відеоадаптер, адаптери клавіатури й маніпулятора типу «миша», жорсткий диск збільшеної місткості. Крім того, можна встановити другий адаптер мережі Ethernet для під'єднання системи до локальної або глобальної мережі. Показаний на рис. 2.11 послідовний інтерфейс RS-232 призначений для технічного обслуговування системи. Кожний концентратор високошвидкісної мережі (Router) може обслуговувати до восьми HS-Link каналів.

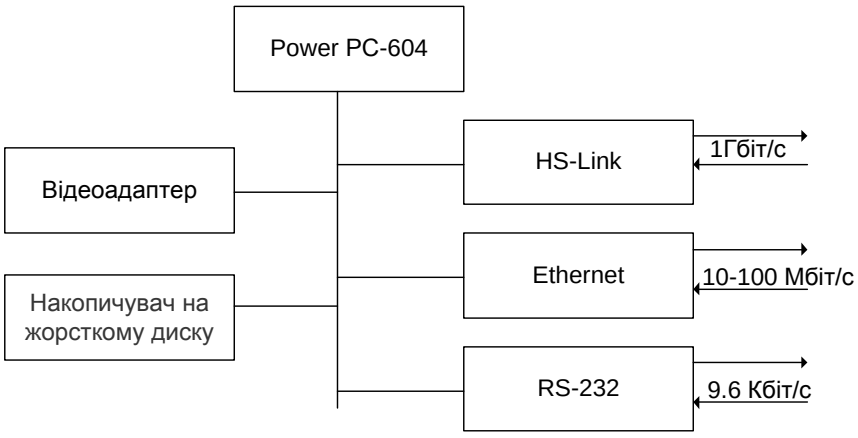


Рис. 2.11. Структура процесорного вузла системи Parsytec CC

Системи цієї серії мають такі особливості:

– висока швидкість передачі даних між процесорами за рахунок застосування спеціальних високошвидкісних комунікаційних модулів і лінків;

– кожний обчислювальний вузол обслуговує власна копія операційної системи, тобто кожний вузол є незалежною однопроцесорною обчислювальною системою зі своєю дисковою пам’яттю;

– для функціонування системи не потрібно наявності додаткової машини для керування, оскільки вона входить до локальної Ethernet-мережі, яка вже має визначені принципи керування під час свого функціонування;

– зв’язок між вузлами здійснюється як через стандартну внутрішню Ethernet-мережу, так і через спеціальну високошвидкісну мережу HS-Link.

Системи на основі розподіленої пам’яті є масштабованими. Для об’єднання великої кількості процесорів може використовуватися легко масштабована, необмежено нарощувана топологія, наведена на рис. 2.12.

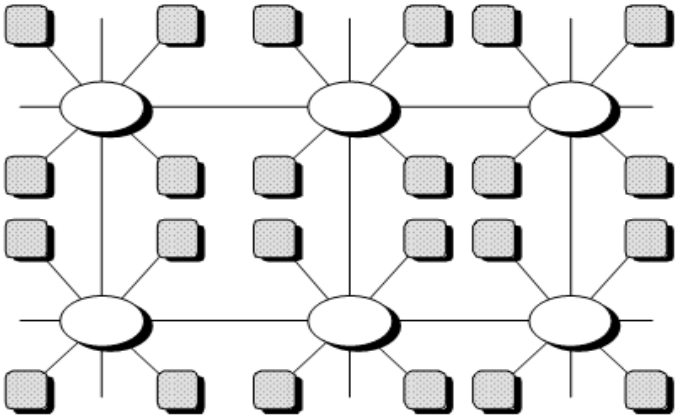


Рис. 2.12. Масштабований варіант топології системи Parsytec CC

Ще однією особливістю систем такого типу є можливість їх побудови без концентраторів високошвидкісної мережі (Routers). Для цього достатньо встановити на кожному із внутрішніх вузлів ланцюжка процесори (рис. 2.13) програмне забезпечення адаптера HS-Link.

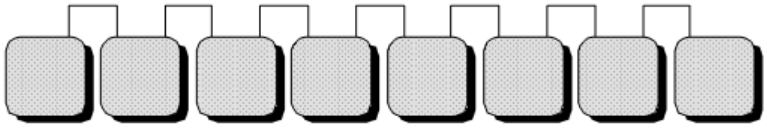
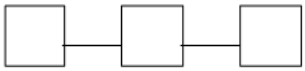


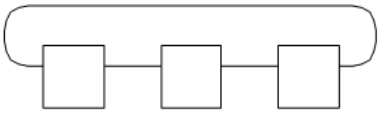
Рис. 2.13. Варіант топології системи Parsytec CC, що не вимагає використання високошвидкісних комутаторів

2.3. Топології багатопроцесорних систем

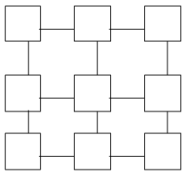
Під топологією багатопроцесорної системи розуміємо спосіб з'єднання процесорних вузлів між собою каналами передачі даних. Зручно подати топологію системи у вигляді графу, вершини якого відповідають процесорним вузлам, а ребра – каналам зв'язку відповідно. Умовно топології можна поділити на фіксовані, реконфігуровані, з одного боку, на регулярні й нерегулярні – з другого. Серед регулярних широко використовують топології таких типів:



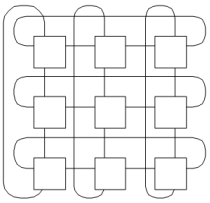
Топологія «лінійка»



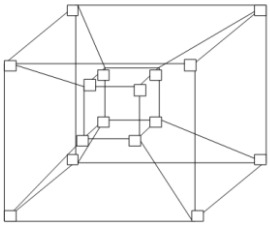
Топологія «кільце»



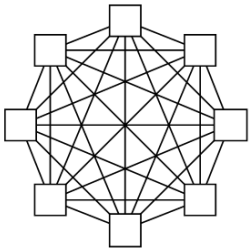
Топологія «решітка3х3»



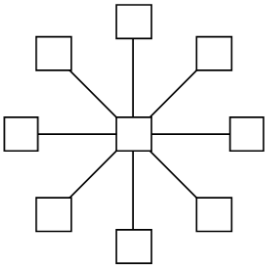
Топологія «тор 3х3»



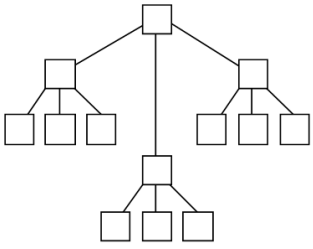
Топологія «гіперкуб ступеня 4»



Топологія «клику»



Топологія «зірка»



Топологія «трійкове дерево»

Властивості використовуваної топології визначають лише ефективність виконання паралельної програми, але й можливість масштабування самої обчислювальної системи. Будь-яку кількість процесорів можна об'єднати в топології типу «лінійка», «кільце», «кліку». Однак для побудови топологій типу «решітка» або «тор» потрібно $n1 \cdot n2$ процесорів, а тому збільшення кількості процесорів можливе тільки квантами розміру $n1$ або $n2$. Для побудови топології «гіперкуб» потрібно $n2$ процесорів, це означає, що кожна наступна система має містити вдвічі більше процесорів, ніж попередня; така топологія вимагає для своєї реалізації наявності n каналів зв'язків на кожному процесорному вузлі, що також обмежує можливості збільшення кількості вузлів у системі.

Щоб підвищити ефективність виконання програм на обчислювальних системах, необхідно узгоджувати фізичну топологію системи й топологію задачі. Велика частина задач математичної фізики успішно розв'язується на системах, процесори яких об'єднані за топологією «решітка». Прямокутні просторові сітки, використовувані для чисельного інтегрування систем диференціальних рівнянь, які описують такі задачі, зручно поділяти на прямокутні частини, які безпосередньо відбиваються на решітці процесорів. Важливо, що від фізичної топології може суттєво залежати ефективність виконання конкретної програми. Перші багатопроцесорні системи, які набули поширення, мали обмежені можливості з погляду реконфігурування. Системи на основі трансп'ютерів дозволяли будувати топології процесорів «двовимірна решітка», «кільце», «циліндр» і «тор». Однак системами з топологією «тор» те, що можна було побудувати на основі трансп'ютерів, можна назвати умовно: для реалізації топології «тор» потрібно саме чотири *link*, при цьому не залишається *link* для приєднання системи до обчислювальної машини, яка керує обчислювальним процесом (рис. 2.14).

У разі такого поєднання процесорів порушується однорідність процесів передачі даних між ними під час обчислень. Побудувати таким чином топологію «гіперкуб» розміром більше 16 ($2^4 = 16$) процесорів, або топологію «тривимірний куб», важко, не згрупувавши процесори у блоки, наприклад,

поєднавши на основі спільної пам’яті у блоки по два процесори, отримаємо обчислювальні вузли з вісьмома *link* (рис. 2.15).

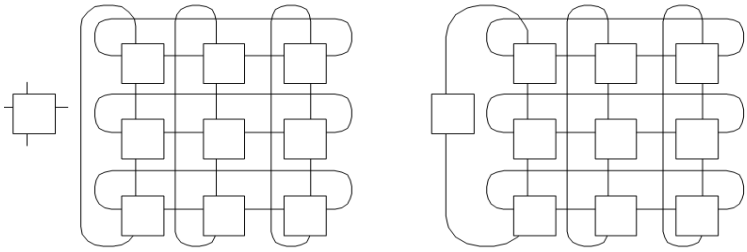


Рис. 2.14. Під’єднання процесорів, з’єднаних за топологією «тор», до керувальної машини

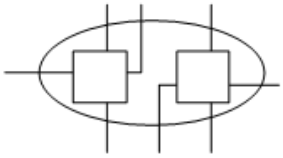


Рис. 2.15. Об’єднання процесорів у обчислювальні вузли

Гіперкуб і тривимірний куб містять топологію «решітка». У зв’язку з чим виникає запитання, якою має бути фізична топологія, щоб за наявності досить жорстких обмежень щодо кількості каналів отримати мінімальну відстань між найбільш віддаленими процесорами? Інакше кажучи, як можна мінімізувати діаметр графу процесорів, зберігаючи низку зв’язків, необхідних для ефективного виконання програми?

Можливе часткове розв’язання цієї задачі – побудова топологій типу «піраміда» (рис. 2.16, 2.17), які дають найкращий з можливих результат, але не є масштабованими.

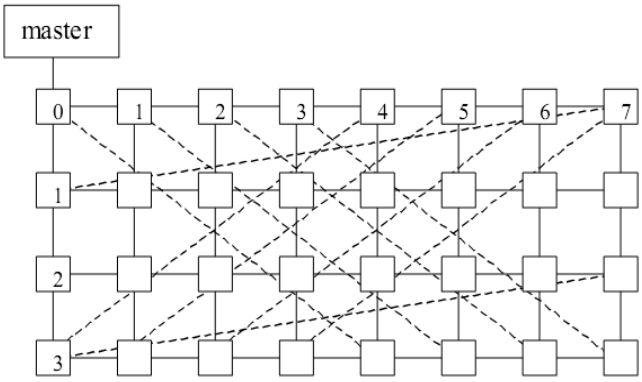


Рис. 2.16. Приклад графу із 32 процесорами з діаметром і радіусом, що дорівнюють чотирьом

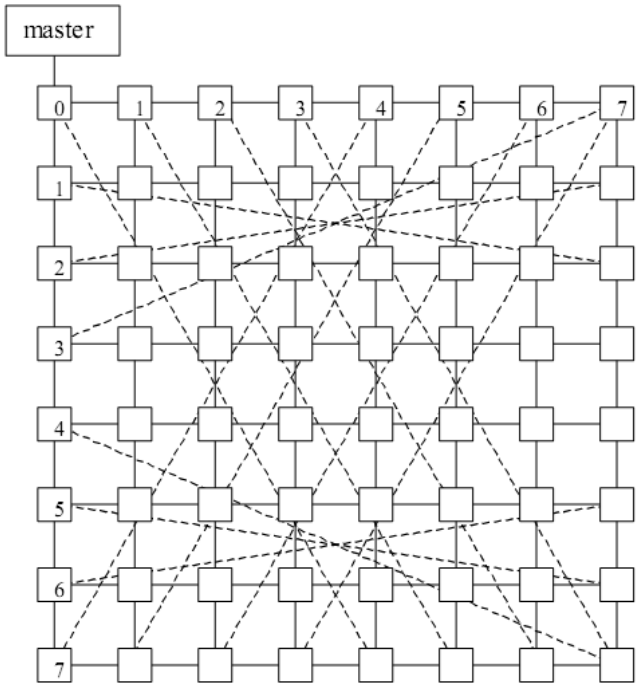


Рис. 2.17. Приклад графу «піраміда» із 64 процесорами з діаметром і радіусом, що дорівнюють шести

2.4. Концепції програмних рішень

Найважливішою властивістю розподіленої системи є можливість здійснення ефективного обміну даними під час взаємодії програмних компонентів, які містяться на одному або на різних комп'ютерах.

Розподілені системи подібні до традиційних операційних систем. Вони, по-перше, працюють як менеджери ресурсів (resource managers) наявного апаратного забезпечення, чим допомагають багатьом користувачам і прикладним програмам спільно використовувати такі ресурси, як процесори, пам'ять, периферійні пристрої, мережу й дані всіх видів. По-друге, розподілена система приховує складність і гетерогенну природу апаратного забезпечення, на основі якого її створено, надаючи віртуальну машину для виконання прикладних завдань.

Першорядними програмними компонентами розподілених систем є операційні системи й системи проміжного рівня. Основні дані щодо розподілених і мережних операційних систем та засобів проміжного рівня подано у табл. 2.1.

Таблиця 2.1. Короткий опис розподілених і мережних операційних систем та засобів проміжного рівня

Система	Опис	Основне призначення
Розподілені операційні системи	Сильнозв'язні операційні системи для мультипроцесорів і гомогенних мультикомп'ютерних систем	Приховання й керування апаратним забезпеченням
Мережні операційні системи	Слабкозв'язні операційні системи для гетерогенних мульти-комп'ютерних систем (локальних або глобальних мереж)	Надання локальних служб віддаленим клієнтам
Засоби проміжного рівня	Додатковий рівень понад мережною операційною системою, що реалізує служби загального призначення	Забезпечення прозорості розподілу

2.4.1. Операційні системи й розподіленість

Операційні системи (ОС) для розподілених комп'ютерів можна поділити на дві категорії: сильнозв'язні та слабкозв'язні системи. У сильнозв'язних системах операційна система переважно працює з одним, глобальним уявленням ресурсів, якими вона керує. Слабкозв'язні системи можуть являти собою набір операційних систем, кожна з яких працює на власному комп'ютері, але вони функціонують спільно, роблячи власні служби доступними для інших.

Сильнозв'язні операційні системи зазвичай називають розподіленими операційними системами (Distributed Operating System, DOS) і використовують для керування мультипроцесорними й гомогенними мультикомп'ютерними системами. Як і в традиційних однопроцесорних операційних системах, основна мета розподіленої операційної системи полягає у прихованні тонкощів керування апаратним забезпеченням, яке одночасно використовує низку обчислювальних процесів.

Слабкозв'язні мережні операційні системи (Network Operating Systems, NOS) використовують для керування гетерогенними мульти-

комп’ютерними системами. Хоча керування апаратним забезпеченням і є основним завданням мережних операційних систем, ці системи відрізняються від традиційних тим, що локальні служби мають бути доступними для віддалених клієнтів. Мережні ОС надають локальні служби в розпорядження віддаленим клієнтам, тому ці ОС мають сервіс віддаленого доступу – RAS (Remote Access Service).

Розрізняють два типи розподілених операційних систем: мультипроцесорну операційну систему (multiprocessor operating system), яка керує ресурсами мультипроцесора, та мультикомп’ютерну операційну систему (multicomputer operating system), яку розроблено для гомогенних мультикомп’ютерів.

Функціональність розподілених операційних систем зазвичай не відрізняється від функціональності традиційних операційних систем, призначених для комп’ютерів з одним процесором, за винятком того, що вона підтримує роботу декількох процесорів.

Операційні системи традиційно будували для керування комп’ютерами з одним процесором, тому їх називано однопроцесорними. Основним завданням цих систем була організація легкого доступу користувачів і прикладних програм до поділюваних ними пристроїв, таких як процесор, пам’ять, диски й периферійні пристрої. Поділ ресурсів означає можливість використання одного й того ж апаратного забезпечення різними прикладними програмами ізольовано одна від одної. Для прикладної програми це виглядає так, немов ці ресурси перебувають у його повному розпорядженні, при цьому в одній системі може виконуватися одночасно декілька прикладних програм, кожна зі своїм набором ресурсів. У такому разі операційна система реалізує віртуальну машину (virtual machine), надаючи прикладним програмам засоби мультизадачності. За умови спільного використання ресурсів у такій віртуальній машині прикладні програми відділено одну від другої, тому неможливі ситуації, коли під час одночасного виконання двох прикладних програм, наприклад *A* та *B*, програма *A* може змінити дані програми *B* через те, що вона працює з тією самою частиною загальної пам’яті, де ці дані зберігаються. Крім того, прикладні програми мають використовувати надані засоби лише так, як запропоновано операційною системою, натомість операційна система надає

первинні операції зв’язку, які можна використовувати для пересилання повідомлень між прикладними програмами на різних машинах.

Операційна система має повністю контролювати використання й розподілення апаратних ресурсів, тому більшість процесорів підтримують як мінімум два режими роботи: у режимі ядра (kernel mode) та у режимі користувача (user mode). У режимі ядра виконуються всі дозволені інструкції, при цьому в ході виконання доступні вся наявна пам’ять і будь-які регістри. У режимі користувача доступ до регістрів й пам’яті обмежений, тобто прикладні програми не можуть працювати з пам’яттю за межами набору адрес, установлених для них операційною системою, або звертатися прямо до регістрів пристроїв. На час виконання коду операційної системи процесор перемикається в режим ядра. Єдиний спосіб перейти з режиму користувача в режим ядра – це зробити системний виклик, реалізований через операційну систему. Оскільки системні виклики здійснюють лише базові служби, які надає операційна система, а обмеження доступу до пам’яті й регістрів нерідко реалізується апаратно, то операційна система може повністю їх контролювати.

Наявність двох режимів роботи зумовило таку організацію операційних систем, за якої майже весь їх код виконується в режимі ядра, в результаті чого часто створюють гігантські монолітні програми, які працюють у єдиному адресному просторі. Такий підхід ускладнює процес переналаштування системи, оскільки замінити або адаптувати компоненти операційної системи без повного перезавантаження, а можливо, й повної перекомпіляції й нової установки дуже важко. З погляду вимог до проектування програм, таких як відкритість, надійність або легкість обслуговування, монолітні операційні системи неефективні.

Більш зручна організація операційної системи у вигляді двох частин. Одна частина містить набір модулів для керування апаратним забезпеченням, який легко може виконуватися в режимі користувача. Наприклад, керування пам’яттю полягає передусім у відстеженні, які блоки пам’яті виділені під процеси, а які вільні, причому робота в режимі ядра необхідна під час установлення регістрів блока керування пам’яттю.

Друга частина операційної системи має невелике мікроядро (microkernel), яке містить винятково код, що виконується в режимі ядра. На практиці

мікроядро має містити лише код для установлення регістрів пристроїв, перемикання процесора з одного процесу на другий, роботи з блоком керування пам’яттю й перехоплення апаратних переривань. Крім того, у ньому зазвичай міститься код, який перетворює виклики відповідних модулів рівня користувача операційної системи в системні виклики й повертає результати. Такий підхід зумовлює організаціїю ОС, показану на рис. 2.18.

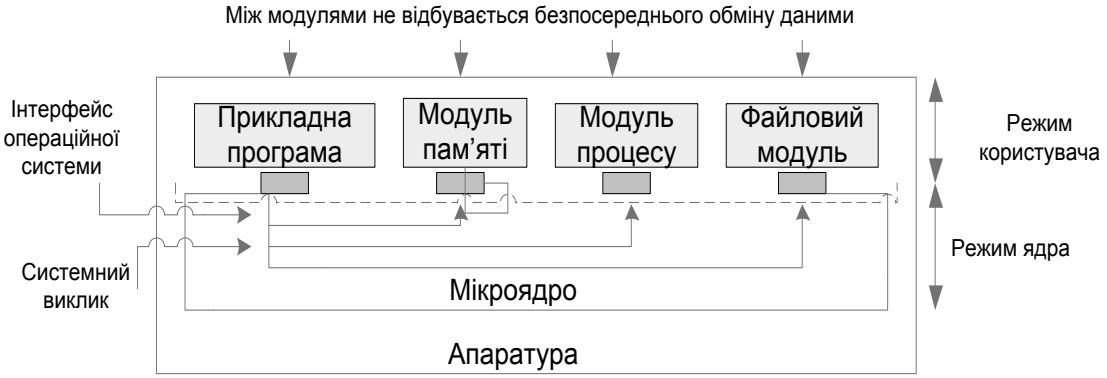


Рис. 2.18. Організація операційної системи з використанням мікроядра

Використання мікроядра має низку переваг, найбільш важливою з яких є гнучкість системи, оскільки більша частина операційної системи виконується в режимі користувача, то порівняно просто замінити один з модулів без повторних компіляції або установлення всієї системи. Інша суттєва перевага полягає в тому, що модулі рівня користувача можуть розміщуватися на різних машинах. Підхід з використанням мікроядра зручний для роботи однопроцесорних операційних систем на розподілених комп'ютерах.

У мікроядер є два істотні недоліки: по-перше, вони працюють інакше, ніж наявні операційні системи; по-друге, мікроядро вимагає додаткового обміну даними, що знижує продуктивність.

Важливим подальшим кроком розвитку однопроцесорних операційних систем є можливість підтримки декількох процесорів, що мають доступ до спільно використовуваної пам’яті. Такі операційні системи називають мультипроцесорними. Всі структури даних, необхідні операційній системі для підтримки апаратури, зокрема декількох процесорів, розміщуються в пам’яті, що доступна декільком процесорам, тому мають бути захищені від паралельного доступу для забезпечення їх цілісності.

Багатопроцесорні операційні системи потрібні для підтримання високої продуктивності конфігурацій з декількома процесорами. Основне їх завдання – забезпечити прозорість кількості процесорів для прикладних програм. Повідомлення, якими обмінюються різні прикладні програми або їх частини, вимагає тих самих примітивів, що застосовують багатозадачні однопроцесорні операційні системи, тобто всі повідомлення працюють з даними у спеціальній, області пам'яті, яка спільно використовується, тому необхідно захистити дані від одночасного доступу до них за допомогою примітивів синхронізації, зокрема двох найбільш важливих (і еквівалентних) примітивів –семафорів і моніторів.

Семафор (semaphore) можна подати у вигляді цілого числа, що підтримує дві операції: *up* (збільшити) і *down* (зменшити). У разі зменшення числа спочатку перевіряється, чи перевищує значення семафора нульове значення. Якщо це так, його значення зменшується й виконання процесу триває; якщо ж значення семафора нульове, то процес, який виконує виклик, блокується. Оператор збільшення виконує протилежну дію: спочатку він перевіряє всі заблоковані в цей час процеси, які не завершилися під час виконання попередньої операції зменшення. Якщо такі наявні, то він розблоковує один з них і продовжує роботу, інакше він просто збільшує лічильник семафора. Процес, який розблоковано, виконується до виклику операції зменшення. Важливою властивістю операцій з використанням семафорів є те, що вони атомарні (atomic), тобто коли запущено операцію зменшення або збільшення до моменту її завершення (або до моменту блокування процесу), жодний інший, процес не може отримати доступ до семафора.

Використання семафорів для синхронізації процесу спричиняє багато похибок, за винятком простого захисту поділюваних даних. Альтернативою семафорам у багатьох сучасних системах, що підтримують паралельне програмування й надають бібліотеки для його реалізації, є використання моніторів.

Формально **монітор** (monitor) є конструкцією мови програмування, такою самою, як об'єкт в об'єктно-орієнтованому програмуванні. Монітор можна розглядати як модуль, що містить змінні та процедури, а доступ до змінної можна отримати,лише, викликавши одну з процедур монітора. Таким чином,

монітор дуже схожий на об’єкт, бо також має свої захищені дані, доступ до яких можна одержати лише через методи, реалізовані в цьому об’єкті. Відмінність між моніторами й об’єктами полягає в тому, що монітор дозволяє виконання процедури тільки одному процесу в кожний момент часу. Якщо процедура, яка міститься в моніторі, виконується процесом *A* (*A* ввійшов у монітор), а процес *B* також викликає одну з процедур монітора, то процес *B* буде блоковано до завершення виконання процесу *A* (доки *A* не залишить монітор).

Мультикомп’ютерні операційні системи мають набагато різноманітнішу структуру і складніші, ніж мультипроцесорні. Для мультикомп’ютерних операційних систем структури даних, необхідні для керування системними ресурсами, не мають задовольняти умову їх спільного використання, оскільки їх не потрібно розміщувати у загальній пам’яті. Єдиним можливим видом зв’язку є передача повідомлень (message passing). Мультикомп’ютерні операційні системи здебільшого організовані так, як показано на рис. 2.19.

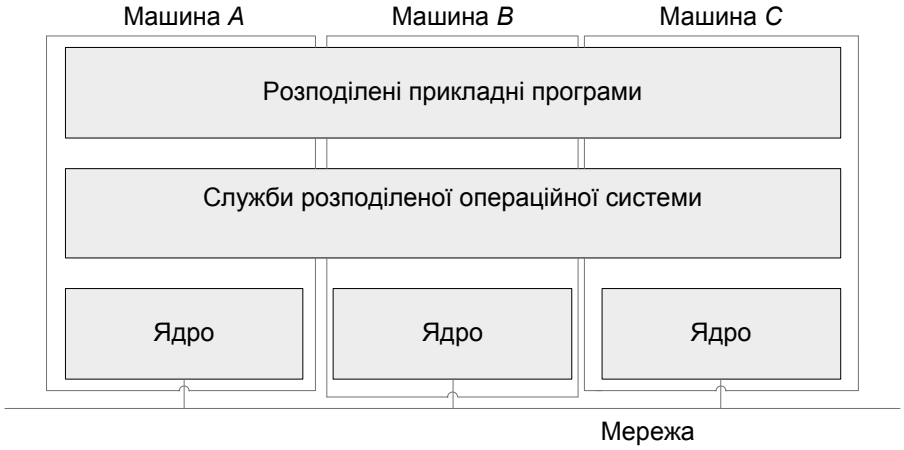


Рис. 2.19. Загальна структура мультикомп’ютерних операційних систем

Кожний вузол системи має своє ядро, яке містить модулі для керування локальними ресурсами (пам’яттю, локальним процесором, локальними дисками та ін.), а також окремий модуль для міжпроцесорної взаємодії, тобто відсилання повідомлень на інші вузли і прийому повідомлень від них. Над кожним локальним ядром міститься рівень програмного забезпечення загального призначення, що реалізує операційну систему у вигляді віртуальної машини, яка підтримує паралельну роботу із різними завданнями. Цей рівень може навіть надавати абстракцію мультипроцесорної машини, тобто повну про-

грамну реалізацію пам’яті, яка спільно використовується. Додаткові засоби необхідні, наприклад, для обрання завдань процесорам, маскування збоїв апаратури, забезпечення прозорості збереження даних і загального обміну між процесами.

Системи з розподіленою поділюваною пам’яттю – це системи, які використовують віртуальну пам’ять кожного окремого вузла для підтримання загального віртуального адресного простору, що зумовлює використання розподіленої поділюваної пам’яті (Distributed Shared Memory, DSM) зі сторінковою організацією. Принцип роботи цієї пам’яті такий: у системі з DSM адресний простір розділено на сторінки (зазвичай по 4 або 8 Кбайт), які розподілено по всіх процесорах системи. Коли процесор адресується до пам’яті, що не є локальною, відбувається внутрішнє переривання. Операційна система зчитує в локальну пам’ять сторінку, що містить зазначену адресу, і перезапускає виконання інструкції, яка спричинила переривання та яка після цього успішно виконується. Як тимчасове сховище інформації використовується не диск, а віддалена оперативна пам’ять.

Одним з покращень базової системи, що підвищує її продуктивність, є реплікація сторінок, які оголошуються закритими для запису, наприклад, таких, що містять текст програми, константи «тільки для читання» або інші закриті на запис структури.

Ще одним покращенням є можливість реплікації також не закритих на запис сторінок, оскільки виконується лише читання, то ніякої різниці між реплікацією закритих і не закритих на запис сторінок немає. Однак, якщо реплікована сторінка змінюється, то необхідно вживати спеціальних заходів для запобігання появі низки несумісних копій. Зазвичай усі копії, крім однієї, перед записуванням вважаються хибними.

Додаткового підвищення продуктивності можна досягти відходом від строгої відповідності між сторінками, які реплікуються, щоб окрема копія тимчасово відрізнялася від інших. Практика показує, що цей підхід насправді може допомогти, але слід обов’язково відслідковувати можливу несумісність. Оскільки основною передумовою розробки DSM була простота програму-

вання, то послаблення відповідності між сторінками не набуває реального застосування.

Ще однією проблемою під час розробки ефективних систем DSM є розмір сторінок. Витрати на передачу сторінки мережею передусім визначаються витратами на підготовку до передачі, а не обсягом переданих даних. Відповідно, великий розмір сторінок може зменшити загальну кількість сеансів передачі у разі необхідності доступу до великої кількості послідовних елементів даних. Натомість, якщо сторінка містить дані двох незалежних процесів, що виконуються на різних процесорах, операційна система буде змушена постійно пересилати цю сторінку від одного процесора до другого, як показано на рис. 2.20. Розміщення даних двох незалежних процесів на одній сторінці називають помилковим поділом (false sharing).

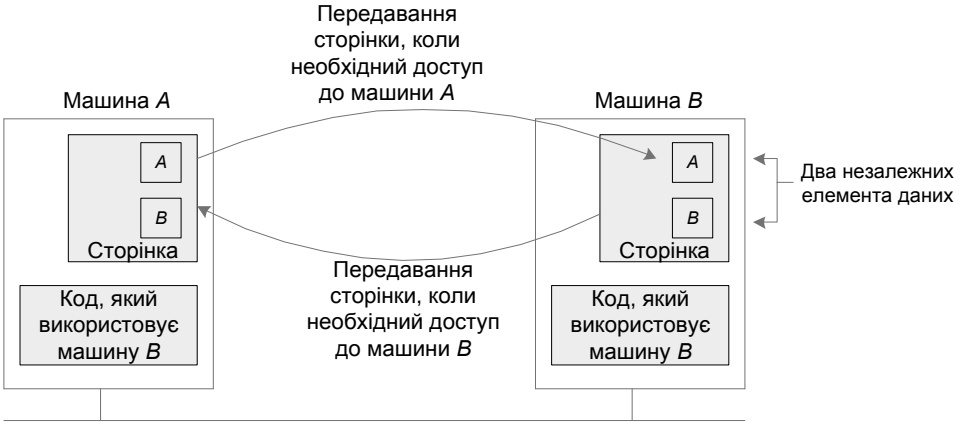


Рис. 2.20. Помилковий поділ сторінки двома незалежними процесами

Для досягнення високої продуктивності великомасштабних мультикомп'ютерних систем здійснюється пересилання повідомлень, незважаючи на високу складність його реалізації порівняно з програмуванням систем з віртуальною пам'ятью спільного використання. Це дозволяє зробити висновок про те, що DSM не є ефективною для високопродуктивного паралельного програмування.

На відміну від розподілених операційних систем, мережні не мають потреби в тому, щоб апаратне забезпечення, на якому вони функціонують, було гомогенним і керованим як єдина система. Мережні системи будуються з набору однопроцесорних систем, кожна з яких має власну операційну сис-

тему, як показано на рис. 2.21. Машини та їх операційні системи можуть бути різними, але їх з'єднано в мережу. Крім того, мережна операційна система дозволяє користувачам використовувати служби, розміщені на конкретній машині, зокрема:

- віддалене з'єднання користувача з другою машиною;
- віддалене копіювання файлів з однієї машини на другу.

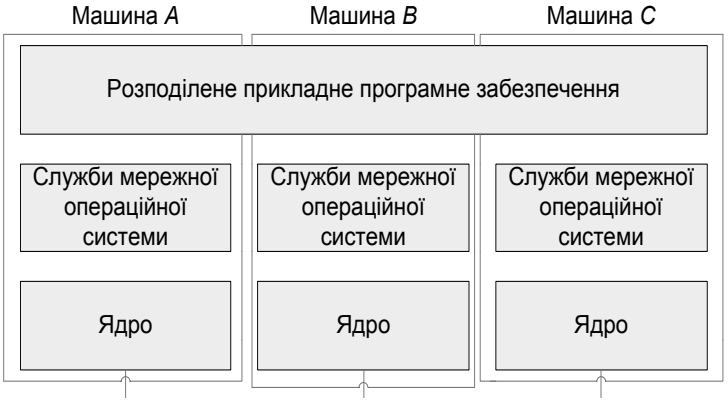


Рис. 2.21. Загальна структура мережної операційної системи

Для віддаленого з'єднання користувача з другою машиною використовується команда *rlogin machin*, у результаті виконання якої відбувається перемикання робочої станції користувача в режим віддаленого терміналу, під'єднаного до віддаленої машини. Команди передаються на віддалену машину, а результати з неї відображаються у вікні на екрані користувача. Щоб переключитися на другу віддалену машину, необхідно відкрити нове вікно і скористатися командою *rlogin* для з'єднання. Вибір виконується вручну.

Приклад. Команда віддаленого копіювання файлів копіює файл *file 1* з машини *machine 1* на *machine 2* і присвоює йому ім'я *file 2*, при цьому переміщення файлів задається явно, й користувачеві необхідно точно знати, де перебувають файли і як виконуються команди.

Така форма зв'язку вкрай примітивна. Більш зручним варіантом зв'язку і спільного використання інформації є один з підходів, що передбачає створення глобальної загальної файлової системи, доступної з усіх робочих станцій. Файлова система підтримується однією або декількома машинами, які називають **файловими серверами** (file servers).

Файлові сервери отримують запити від програм користувачів, що запускаються на інших машинах (не на серверах), які називають **клієнтами** (clients),

для читання й записування файлів. Кожний запит, що надійшов, перевіряється й виконується, а результат пересилається назад, як показано на рис. 2.22.

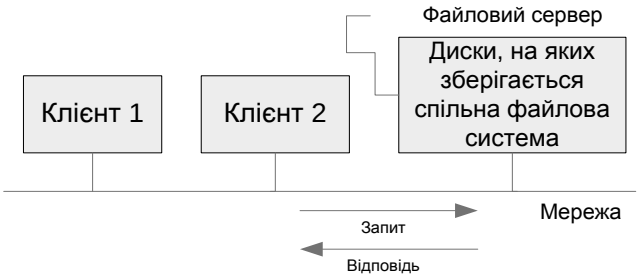


Рис. 2.22. Два клієнти й сервер у мережній операційній системі

Файлові сервери зазвичай підтримують ієрархічні файлові системи, кожна з яких має кореневий каталог, який містить вкладені каталоги й файли. Робочі станції можуть імпортувати або монтувати ці файлові системи, збільшуючи свою локальну файлову систему за рахунок файлової системи сервера.

Мережні операційні системи виглядають значно простіше, ніж розподілені. Основна відмінність між мережними й розподіленими типами операційних систем полягає у тому, що в розподілених операційних системах повністю виконується вимога прозорості системи, тобто подання всього інформаційного простору у вигляді єдиної системи.

Неповне виконання вимоги прозорості в мережних операційних системах має деякі очевидні недоліки. Наприклад, з ними часто складно працювати, оскільки користувач змушений вручну приєднуватися до віддалених машин або копіювати файли з однієї машини на другу, що спричиняє проблеми з керуванням ресурсами всієї системи. Оскільки всі машини, якими керує мережна операційна система, незалежні, то часто й керувати ними можна тільки незалежно, у результаті чого користувач може виконати віддалене з'єднання з машиною *X*, лише зареєструвавшись на ній. Якщо користувач хоче використовувати один пароль для роботи з декількома машинами, то за потреби змінити пароль він змушений буде змінити його на кожній машині. Очевидно, що на кожній машині наявні свої права доступу. Такий децентралізований підхід до безпеки ускладнює захист мережної операційної системи.

Мережні операційні системи порівняно з розподіленими мають також і переваги. Оскільки вузли мережних операційних систем значною мірою не

залежать один від одного, додати або видалити машину дуже легко. Зокрема, щоб додати вузол, треба приєднати відповідну машину до загальної мережі й повідомити про її наявність іншим машинам мережі. В Internet, наприклад, додавання нового сервера відбувається в такий спосіб: щоб відомості про машину потрапили в Internet, необхідно надати їй мережну адресу, а краще символічне ім'я, яке потім буде внесено в доменну систему іменування DNS разом з її мережною адресою.

Domain Name System (DNS) – доменна система іменування (імен), або служба доменних імен, Internet-служба є розподіленою по всій земній кулі базою даних для ієрархічної системи імен мереж і комп'ютерів, під'єднаних до глобальної мережі, а також є способом або протоколом прикладного рівня перетворення рядкових адрес серверів Internet у числові IP-адреси. Протокол DNS працює над протоколом UDP (протоколом дейтаграм користувача, протоколом транспортного рівня з набору протоколів TCP/IP) і йому призначено порт із номером *53*; DNS також часто використовують для розподілу навантаження між дублюючими серверами (дзеркалами) популярних сайтів і поштових серверів.

Система імен у DNS – ієрархічно організована розподілена система з дублюванням усіх функцій між двома й більше серверами (рис. 2.23).

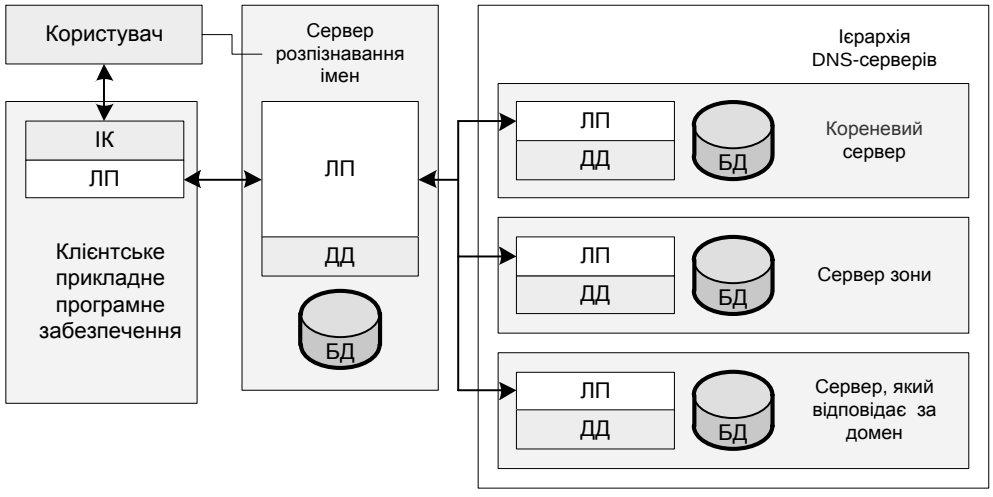


Рис. 2.23. Система DNS: ІК – інтерфейс користувача; ЛП – логіка прикладних програм; ДД – доступ до даних

Запит користувача на перетворення імені (наприклад, www.microsoft.com) у мережну адресу (65.225.182.56) передається серверу розпізнавання імен постачальника послуг Internet, сервер почергово опитує сервери з ієрархії служби імен, починаючи з кореневих серверів, які повертають адреси серверів, що відповідають за зону домена, у яких містяться проміжні шляхи.

Потім опитується сервер, який відповідає за зону (у цьому разі – .com), що повертає адреси серверів, які відповідають за домен других рівнів, і так далі. Сервери імен кешують інформацію про відповідність імен і адрес для зменшення навантаження на систему. Програмне забезпечення на комп'ютері користувача зазвичай може з'єднатися як мінімум з двома різними серверами розпізнавання імен, проте і в системі розпізнавання імен не всі вимоги до розподілених систем виконано, зокрема така система не містить явних механізмів забезпечення безпеки. Це призводить до регулярних атак на сервери імен, щоб вивести їх з ладу, наприклад значною кількістю запитів.

2.4.2. Проміжне середовище

Розподілені операційні системи не призначені для керування набором незалежних комп'ютерів, а мережні операційні системи не створюють уявлення однієї узгодженої системи. Бажаною властивістю розподіленої системи є можливість організації ефективного обміну даними під час взаємодії програмних компонентів, що перебувають на одному або на різних комп'ютерах. Щоб насправді побудувати розподілену систему, служб мережної операційної системи недостатньо, необхідно долучити до них прикладні програмні компоненти для покращення прозорості розподілу. Задовольнити вимоги масштабованості й відкритості можна за допомогою додаткового рівня програмного забезпечення, який у мережних операційних системах дозволяє якось сховати від користувача різнорідність набору апаратних платформ, чим і підвищити прозорість розподілу. Багато сучасних розподілених систем побудовано з використанням цього додаткового рівня, який названо програмним забезпеченням проміжного рівня, платформою розподілу, системою проміжного рівня або проміжним середовищем (middleware).

Велика кількість розподілених прикладних програм передбачає безпосереднє використання програмного інтерфейсу, наявного у мережних операційних систем та інтерфейсів локальних файлових систем. Проблема такого підходу полягає в тому, що наявність розподілу є занадто конкретною. Для забезпечення додаткового абстрагування між прикладною програмою й мережною операційною системою розміщено рівень програмної підтримки, як показано на рис. 2.24, який і називають **проміжним рівнем**.

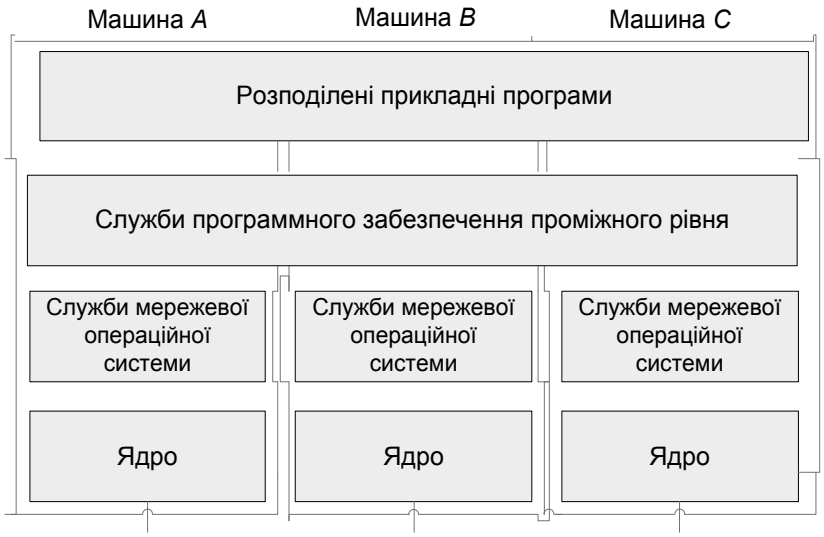


Рис. 2.24. Загальна структура розподілених систем із проміжним рівнем

Система проміжного рівня забезпечує скоординовану роботу мереж і ОС, надаючи можливість використання їх програмного інтерфейсу. Ефективне проміжне середовище має організовувати взаємодію групи комп'ютерів мережі, не порушуючи стек протоколів TCP/IP. Для цього можуть використовуватися системні сокети (unix sockets) у POSIX-системах або іменовані канали (named pipes).

Платформа розподілу підтримує взаємодію між потенційно гетерогенними системами, які на верхньому рівні працюють з прикладними компонентами, може додаватися до локальної операційної системи або отримувати завдання від операційної системи. У такий спосіб підтримується прозорість розподілу. Прикладні програми ізолюються від деталізації структури функцій операційної системи і взаємодії її внутрішніх процесів. Платформи розподілу можуть бути

покладені в основу багатьох конкретних прикладних систем. На платформі розподілу може функціонувати одночасно декілька прикладних програм.

Основне завдання програмного забезпечення проміжного рівня – приховати відмінності базових стандартів ОС від прикладного програмного забезпечення, розробляючи яке, використовують низку моделей або параметрів, що характеризують розподілені системи. Основними моделями є такі, що визначають розподіл і зв'язок.

Крім платформи розподілу, ще однією моделлю організації взаємодії між прикладними програмами або системами в розподіленому гетерогенному середовищі є модель віддаленого виклику процедур RPC (Remote Procedure Call), яка забезпечує виклик процедур з віддалених машин. Під час виклику процедури параметри RPC прозоро передаються на віддалену машину, а результат повертається назад. У разі використання об'єктного підходу прикладні програми створюються за допомогою засобів поєднання розподілених об'єктів, що являють собою певну програмну реалізацію, причому кожний із цих об'єктів реалізує свій інтерфейс, який приховує всі внутрішні деталі реалізації процесу взаємодії від кінцевого користувача. Інтерфейс передає вихідні дані для реалізації методів. Внутрішній та зовнішній процеси, які взаємодіють між собою, бачать лише свій інтерфейс. Часто такі розподілені об'єкти розміщуються на одній машині, а програмне забезпечення проміжного рівня надає дозвіл на доступ до інтерфейсів об'єктів і здійснює цей доступ до них з інших машин. Під час виклику процесором методу інтерфейс перетворює дані, що містяться в інтерфейсі, в повідомлення, яке відсилається об'єкту, який виконує метод і повертає результат.

Прикладну програму, що реалізує кожну із системних функцій платформи розподілу, називають **службою**. Розрізняють такі основні служби платформи розподілу: засоби прозорого доступу до віддалених даних (FS, www); служби віддаленого доступу (для виклику процедур і звертання до розподілених об'єктів); служби іменування (url); засоби збереження даних або засоби живучості (persistence), наприклад розподілені FS, інтегровані бази даних, засоби зв'язування прикладних програм з базами даних; засоби розподілених

транзакцій, які здійснюють низку операцій зчитування й записування у межах однієї атомарної операції.

Програмне забезпечення проміжного рівня дозволяє поєднувати масштабованість і відкритість мережних операційних систем із прозорістю і простотою використання розподілених систем. Таке поєднання дозволяє створювати розподілену інформаційну систему із загальною структурою, показаною на рис. 2.24.

2.5. Поняття розподіленого середовища

Розподілене середовище – віртуальний обчислювальний простір, який може обмежуватися однією розподіленою системою або містити кілька розподілених систем, які взаємодіють між собою. Такий віртуальний обчислювальний простір надається користувачеві у вигляді систематизованого сховища інформаційних та програмних ресурсів, має певну структуру, зрозумілу систему адресації ресурсів та певні моделі обчислювальних процесів або бізнес-процесів цього користувача, які є проблемно-орієнтованими, відповідають певним видам робіт. Користувач звертається до бізнес-процесів, які, у свою чергу, отримують необхідні ресурси для своєї роботи.

Учасниками взаємодії в розподіленому середовищі є окремі сутності, якими можуть бути користувачі, прикладні програми та інші обчислювальні ресурси. Як основу опису взаємодії двох сутностей розглянемо загальну модель взаємодії *клієнт – сервер*, у якій одна зі сторін (клієнт) ініціює обмін даними, надсилаючи запит другій стороні (серверу). Сервер обробляє запит й у разі потреби надсилає відповідь клієнтові (рис. 2.25).

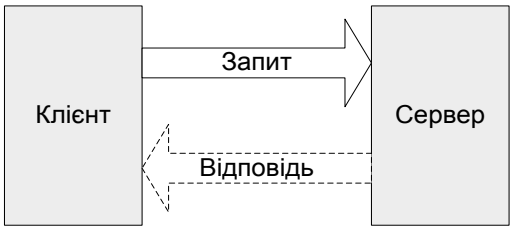


Рис. 2.25. Модель взаємодії *клієнт – сервер*

У базовій моделі *клієнт – сервер* усі процеси в розподілених системах поділяють на дві групи: процеси, які реалізують певну службу, наприклад

службу файлової системи або бази даних, називають **серверами** (servers); процеси, які запитують служби в серверів за рахунок надсилання запиту й очікування відповіді від сервера, називають **клієнтами** (clients). Взаємодія клієнтів і сервера відома також як режим роботи або протокол запит – відповідь (request-reply behavior), зображена на рис. 2.26.

Якщо базова мережа так само надійна, як локальні мережі, то взаємодія між клієнтом і сервером може бути реалізована за допомогою простого протоколу, який не потребує встановлення з'єднання. У цьому разі клієнт, запитуючи службу, перетворює свій запит у форму повідомлення, зазначаючи в ньому службу, якою він бажає скористатися, і необхідні для цього вихідні дані. Потім повідомлення надсилається серверу, який постійно очікує вхідного повідомлення, а отримавши його, обробляє, упаковує результат обробки у відповідне повідомлення й відправляє його клієнтові.

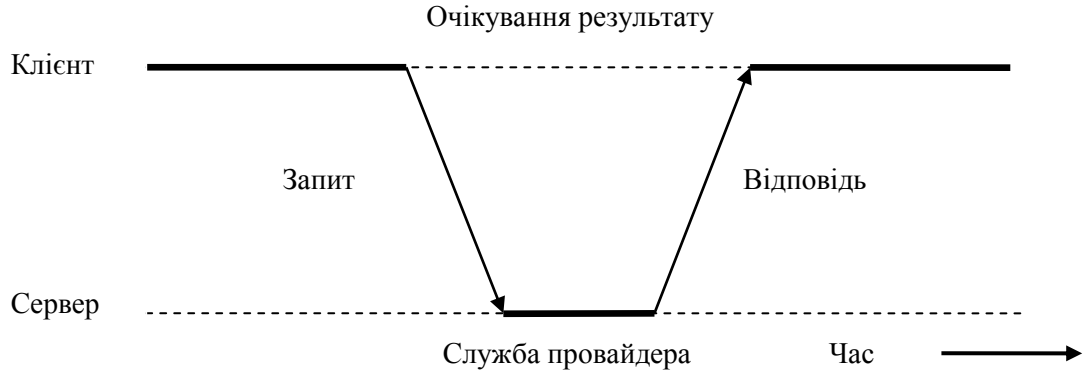


Рис. 2.26. Узагальнена взаємодія між клієнтом і сервером

Використання з'єднання, яке не потребує протоколу, дає істотний виграш в ефективності. До того часу, коли повідомлення не почнуть зникати або ушкоджуватися, можна цілком успішно застосовувати протокол запит – відповідь. На жаль, створити протокол, стійкий до випадкових перебоїв зв'язку, – нетривіальне завдання. Єдине, що можна зробити, – це надати клієнтові можливість повторно надіслати запит, на який не було отримано відповіді. Проблема полягає в тому, що клієнт не може визначити: чи дійсно первинне повідомлення із запитом було загублено чи помилка відбулася під час пере-

дачі відповіді. Якщо втрачено відповідь, то повторне надсилання запиту може призвести до повторного виконання операції.

Взаємодія у межах моделі клієнт – сервер може бути як синхронною, коли клієнт очікує завершення обробки свого запиту сервером, так і асинхронною, коли клієнт надсилає серверу запит і продовжує виконання своєї роботи, не очікуючи відповіді від сервера. Модель клієнта й сервера можна використовувати як основу опису різних способів взаємодії. У цьому контексті важливою є взаємодія складових частин програмного забезпечення, які утворюють розподілену систему.

2.5.1. Розподіл прикладних програм за рівнями

Модель клієнт – сервер потребує розуміння питання, як розподілити програмне забезпечення між клієнтом і сервером. Наприклад, сервер розподіленої бази даних може постійно виконувати роль клієнта, що передає запити на різні файлові сервери, які відповідають за реалізацію таблиць цієї бази даних. У цьому разі сервер баз даних не робить нічого, крім обробки запитів.

Прикладні програми, які створені за технологією клієнт – сервер та призначені для організації доступу користувачів до баз даних, переважно поділяють на три рівні (рис. 2.27): рівень інтерфейсу користувача; рівень обробки; рівень даних.

Рис. 2.27. Логічні рівні прикадної програми

Рівень інтерфейсу користувача містить усе необхідне для безпосереднього спілкування з користувачем, зокрема функції керування дисплеєм, рівень обробки зазвичай містить прикладні програми, а рівень даних – дані, з якими виконується робота.

Рівень інтерфейсу користувача. Рівень інтерфейсу користувача, який зазвичай реалізується на клієнтах, містить програми, за допомогою яких користувач може взаємодіяти з прикладною програмою. Складність програм, що входять до інтерфейсу користувача, досить різна.

Найпростіший варіант програми інтерфейсу користувача, який не містить нічого, крім символьного (не графічного) дисплея, зазвичай використовується у процесі роботи з мейнфреймами. У тому разі, коли мейнфрейм контролює всі процеси взаємодії, включаючи роботу з клавіатурою й монітором, навряд чи можна говорити про використання моделі клієнт – сервер. Однак здебільшого термінали користувачів виконують деяку локальну обробку, здійснюючи, наприклад, еходрук рядків, що вводяться, або надаючи інтерфейс у вигляді форм, у якому можна відредагувати введені дані до їх пересилання на головний комп’ютер. Нині навіть у середовищі мейнфреймів наявні більш досконалі інтерфейси користувачів.

Сучасні інтерфейси користувачів більш функціональні, оскільки вони підтримують спільну роботу прикладних програм через єдине графічне вікно й під час дій користувача забезпечують обмін даними через це вікно. Наприклад, для видалення файлу часто достатньо перенести значок, який відповідає цьому файлу, на значок сміттєвого кошика. Аналогічним способом багато текстових процесорів дозволяють користувачеві переміщувати текст документа в інше місце, користуючись тільки мишею.

Рівень обробки. Багато прикладних програм у моделі клієнт – сервер побудовані з трьох різних частин: частини, яка взаємодіє з користувачем; частини, яка відповідає за роботу з базою даних або файловою системою; і середньої частини, що реалізує основну функціональність прикладної програми та перебуває на рівні обробки. На відміну від інтерфейсів користувача або баз даних, на рівні обробки важко визначити загальні закономірності.

Рівень даних. Рівень даних у моделі клієнт - сервер містить програми, які надають дані прикладним програмам, що їх оброблюють. Специфічною властивістю цього рівня є вимога живучості, тобто коли прикладна програма не працює, дані мають зберігатися в певному місці, оскільки є необхідним їх подальше використання. У найпростішому варіанті рівень даних реалізується файловою системою, але частіше повномасштабною базою даних. У моделі клієнт - сервер рівень даних зазвичай розташований на стороні сервера.

Крім простого зберігання інформації, рівень даних відповідає за підтримку цілісності даних для різних прикладних програм, тобто для бази даних це означає, що метадані (опис таблиць, обмеження і специфічні метадані прикладного програмного забезпечення) зберігаються саме на цьому рівні. Наприклад, якщо у прикладній програмі, яка входить до банківської системи, необхідно сформувати повідомлення щодо боргу клієнта за кредитною карткою, то це може бути зроблено за допомогою тригера бази даних, який у потрібний момент активізує процедуру, яка дозволяє виконати таку дію.

Найчастіше рівень даних формується у вигляді реляційної бази даних. У такому разі ключовою вимогою є незалежність даних, які організуються незалежно від прикладної програми так, щоб зміни в організації даних не впливали на прикладну програму, а прикладна програма не впливала на організацію даних. Використання реляційних баз даних у моделі клієнт - сервер допомагає відокремити рівень обробки від рівня даних, розглядаючи обробку й дані незалежно один від одного.

Однак найбільш поширеними є такі прикладні програми, для яких реляційні бази даних не є найкращим вибором. Характерною рисою таких прикладних програм є робота зі складними типами даних, які простіше моделювати в поняттях об’єктів, а не відношень. До таких типів даних належать дані, які описують об’єкти різної складності: від простих наборів прямокутників і кіл до проекту літака в системах автоматизованого проектування. Мультимедійним системам також значно простіше працювати з відео- та аудіопотоками, використовуючи специфічні для них операції, ніж з моделями цих потоків у вигляді реляційних таблиць.

У тих випадках, коли операції з даними значно простіше виразити в поняттях роботи з об’єктами, має сенс реалізувати рівень даних засобами об’єктно-орієнтованих баз даних, які не тільки підтримують організацію складних даних у формі об’єктів, але і зберігають реалізації операцій над цими об’єктами. Таким чином, частина функціональності, що перебувала на рівні обробки, мігрує на рівень даних.

2.5.2. Варіанти архітектури клієнт - сервер

На практиці різних користувачів розподіленої системи цікавить доступ як до одних і тих самих даних, так і до різних наборів даних. Найбільш простим рознесенням функцій такої системи між декількома комп’ютерами буде поділ логічних рівнів прикладної програми між однією серверною частиною прикладної програми, яка відповідає за доступ до даних, і клієнтськими частинами, що перебувають на декількох комп’ютерах та реалізують інтерфейс користувача. Програмне забезпечення, яке реалізує логіку прикладної програми, може бути віднесене до сервера, клієнтів або розділене між ними (рис. 2.28).

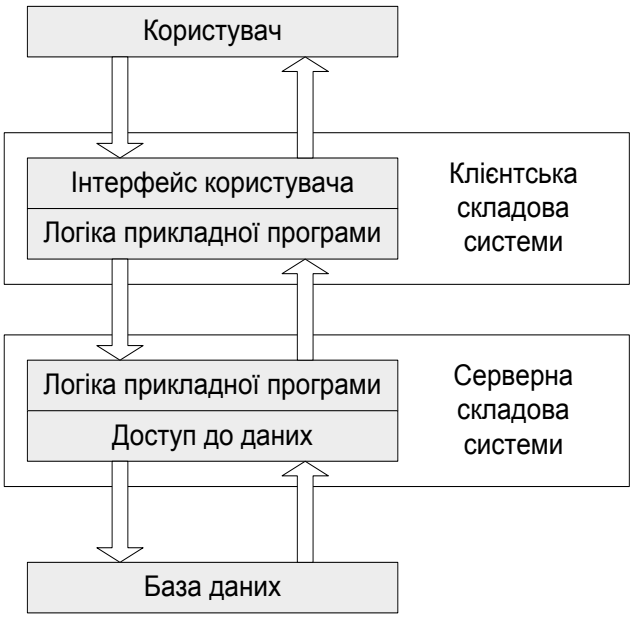


Рис. 2.28. Дволанкова архітектура

Архітектуру прикладного програмного забезпечення, побудовану за таким принципом, називають клієнт серверною або дволанковою. На практиці подібні системи не завжди відносять до класу розподілених, але формально їх можна вважати найпростішою реалізацією розподілених систем.

Наступним етапом розвитку архітектури клієнт - сервер є триланкова архітектура, у якій інтерфейс користувача, логіка прикладної програми й доступ до даних відокремлено в самостійні складові системи, які можуть працювати на незалежних комп'ютерах (рис. 2.29).

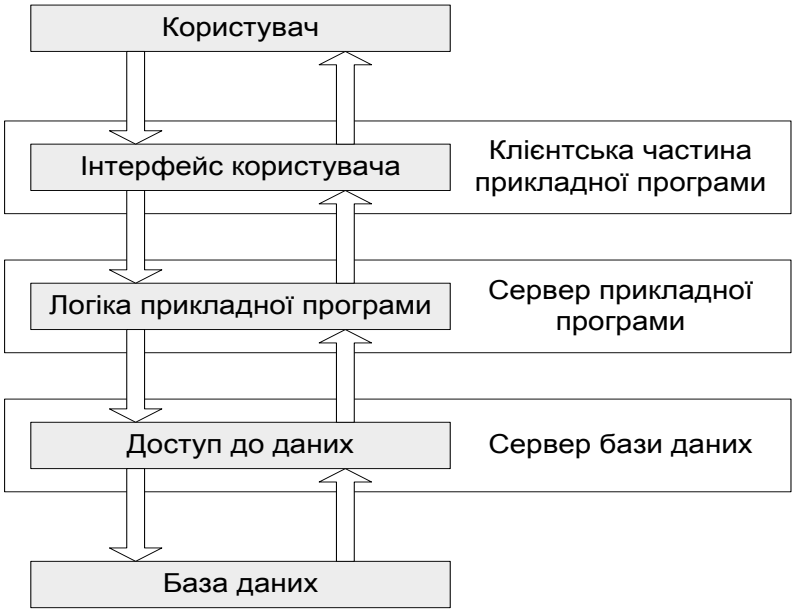


Рис. 2.29. Триланкова архітектура

Запит користувача в таких системах послідовно обробляється клієнтською частиною системи, сервером логіки прикладної програми й сервером баз даних. Однак зазвичай під розподіленою системою розуміють системи з більш складною архітектурою, ніж триланкова.

Розподіленим прикладним програмним забезпеченням, що автоматизує діяльність підприємства або організації, називають системи, логіка прикладних програм яких розподілена між декількома компонентами системи, кожен з яких можна виконувати на окремому комп'ютері.

Приклад. Реалізація логіки прикладної програми системи роздрібної торгівлі має виконувати запити до логіки прикладної програми третіх фірм, зокрема постачальників товарів, систем електронних платежів або банків, що надають споживчі кредити (рис. 2.30).

Таким чином, на практиці під розподіленою системою часто розуміють розширення багатоланкової архітектури, коли запити користувача не проходять послідовно від інтерфейсу користувача до єдиного сервера баз даних.

Приклад. Мережі прямого обміну даними між клієнтами (peer-to-peer networks). Якщо попередній приклад мав «деревоподібну» архітектуру програмного забезпечення, то мережі прямого обміну організовані складніше (рис. 2.31). Подібні системи нині є одними з найбільших серед поширених розподілених систем, що поєднують мільйони комп'ютерів.

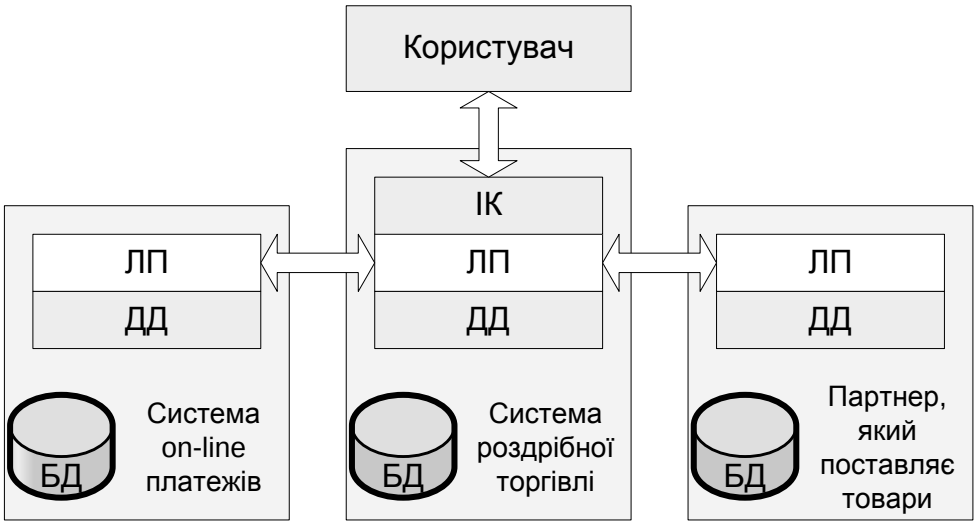


Рис. 2.30. Розподілена система роздрібних продажів

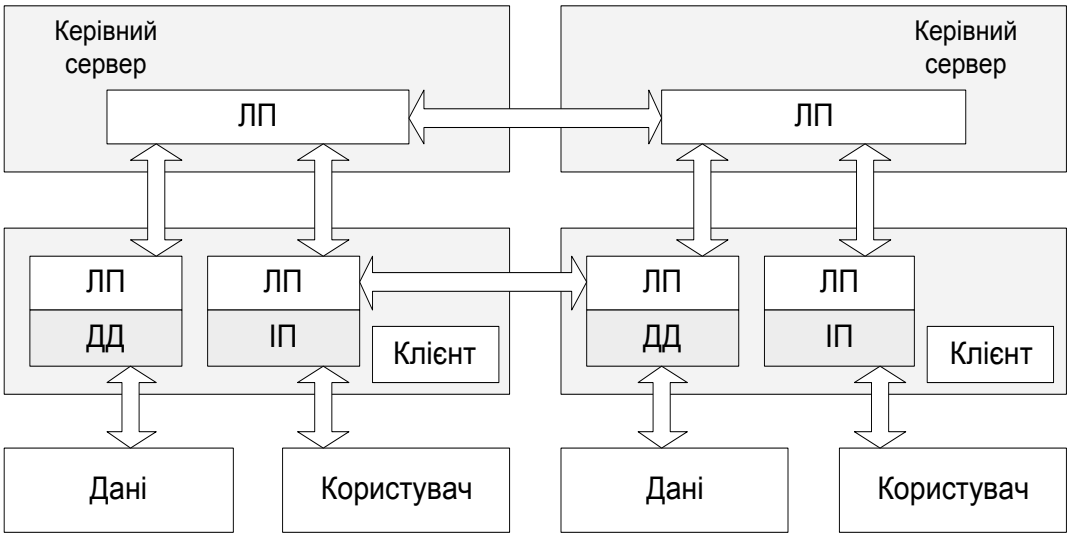


Рис. 2.31. Система прямого обміну даними між клієнтами

Багатоланкові архітектури. Один з підходів до організації взаємодії клієнтів і серверів – це розподіл програм, що перебувають на рівні приклад-

ного програмного забезпечення, який полягає у тому, щоб помістити на клієнтську сторону лише термінальну частину інтерфейсу користувача, як показано на рис. 2.32*а*, дозволивши прикладній програмі віддалено контролювати подання даних. Альтернативою цьому підходу буде передача клієнтові всієї роботи з інтерфейсом користувача (рис. 2.32, *б*). В обох випадках відокремлюється від прикладної програми графічний зовнішній інтерфейс, який є пов'язаним з іншою частиною прикладної програми, що перебуває на сервері, за допомогою специфічного для цієї прикладної програми протоколу. В такій моделі зовнішній інтерфейс виконує лише те, що необхідно для надання інтерфейсу прикладній програмі.

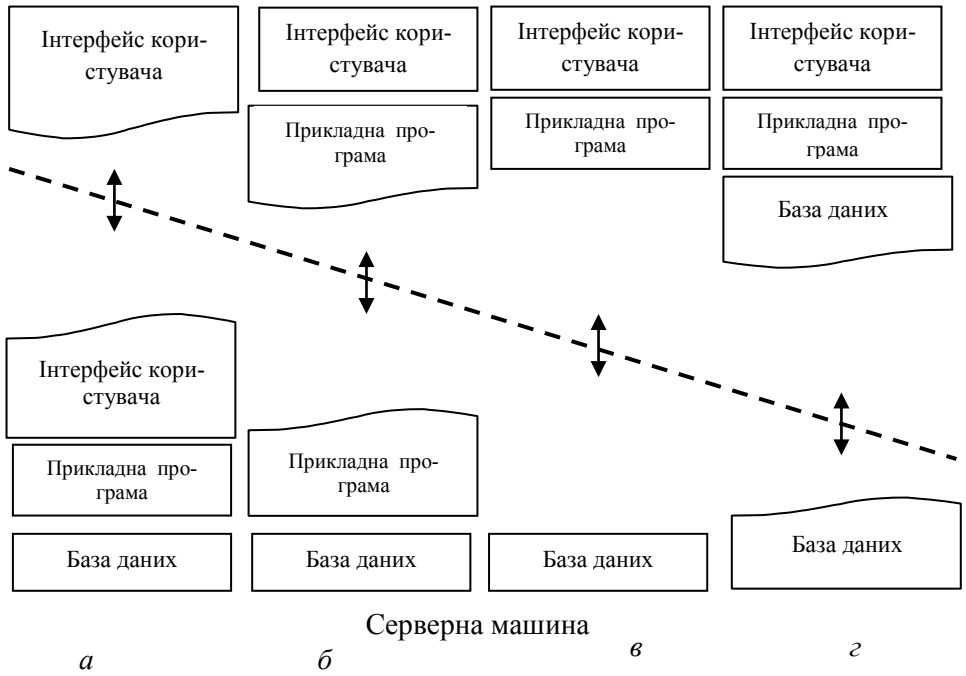


Рис. 2.32. Альтернативні форми організації архітектури клієнт – сервер

У багатьох системах клієнт-сервер поширені типи організації, зображені на рис. 2.32*в* і рис. 2.32*г*, які застосовують у разі, коли клієнтська машина (персональний комп'ютер або робоча станція) з'єднана мережею з розподіленою файловою системою або базою даних. Більша частина прикладних програм працює на клієнтській машині, а всі операції з файлами або базою даних передаються на сервер. Рис. 2.32*г* зображає ситуацію, коли частина даних утримується на локальному диску клієнта. Наприклад, у процесі роботи в

Internet клієнт може поступово створити на локальному диску величезний кеш найвідвідуваніших web-сторінок.

Розглядаючи розподілення програмного забезпечення на клієнтське і серверне, слід враховувати те, що у сервера іноді виникає потреба працювати як клієнт. У такій ситуації (рис. 2.33) маємо фізично триланкову архітектуру (physically three-tiered architecture), у котрій програми, що становлять частину рівня обробки, виносяться на окремий сервер, але прикладні програми можуть частково перебувати й на машинах клієнтів і серверів. Типовим прикладом триланкової архітектури є обробка транзакцій, за якою окремий процес, монітор транзакцій, координує роботу всіх транзакцій.

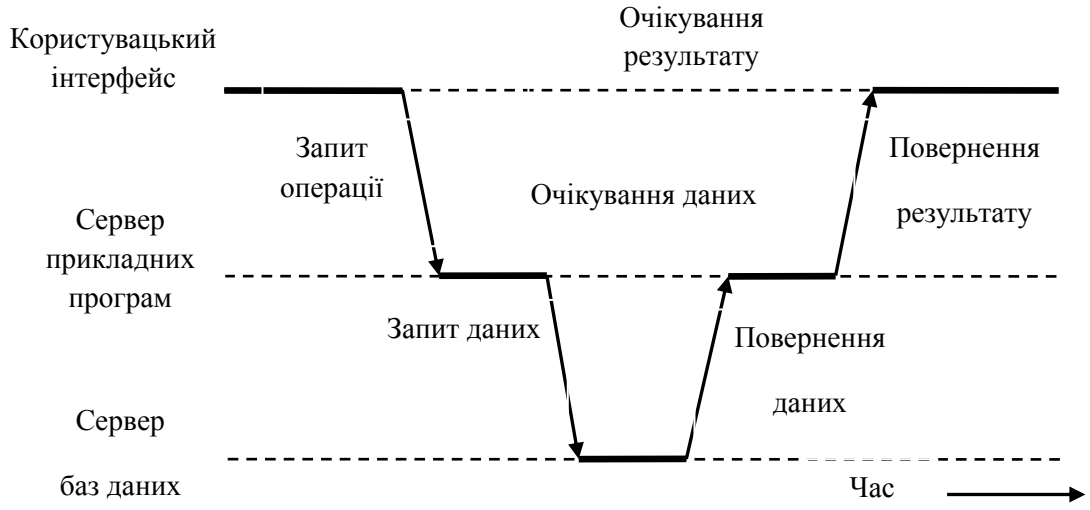


Рис. 2.33. Приклад сервера, що діє як клієнт

Сучасні варіанти архітектури. Багатоланкові архітектури клієнт – сервер є продовженням розподілу прикладних програм на рівні інтерфейсу користувача, компонентів обробки й даних. Різні ланки взаємодіють відповідно до логічної організації прикладних програм. У прикладному програмному забезпеченні розподілена обробка еквівалентна організації багатоланкової архітектури прикладної програми клієнт - сервер. Такий тип розподілу називають **вертикальним** (vertical distribution). Його особливістю є те, що він досягається розміщенням логічно різних компонентів на різних машинах. Це поняття пов’язане з концепцією **вертикальної фрагментації** (vertical fragmentation), яку використовують у розподілених реляційних базах даних, де під цим термі-

ном розуміють розбиттям на стовпці таблиць для їх зберігання на різних машинах.

Вертикальний розподіл – це лише один з можливих способів організації клієнт – серверного прикладного програмного забезпечення. У сучасних архітектурах розподіл на клієнти й сервери відбувається способом, відомим як **горизонтальний розподіл** (horizontal distribution), за якого клієнт або сервер може містити фізично розподілені частини логічно однорідного модуля, причому робота з кожною із частин може виконуватися незалежно для вирівнювання завантаження.

Як найбільш поширений приклад горизонтального розподілу розглянемо web-сервер, реплікований на кілька машин локальної мережі, як показано на рис. 2.34. На кожному із серверів утримується однаковий набір web-сторінок, і щоразу, коли одна з web-сторінок обновляється, її копії негайно розсилаються на всі сервери. Сервер, якому буде переданий вхідний запит, вибирається за правилом «каруселі» (round-robin). Такий варіант горизонтального розподілу досить успішно використовується для вирівнювання навантаження на сервери популярних web-сайтів.

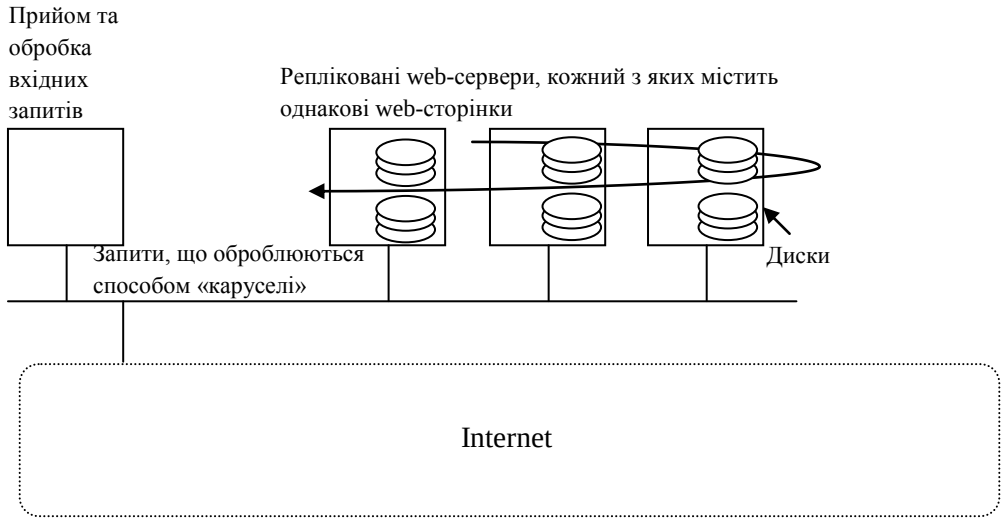


Рис. 2.34. Приклад горизонтального розподілу web-служби

У такий самий спосіб, хоч і менш очевидно, можуть бути розподілені й клієнти. Для нескладного прикладного програмного забезпечення, призначеного для колективної роботи, можна не мати сервера взагалі – це одноранговий розподіл (peer-to-peer distribution), що відбувається, наприклад, якщо

користувач хоче зв’язатися з іншим користувачем і обидва повинні запустити одну прикладну програму, щоб розпочати сеанс. Третій клієнт може також спілкуватися з одним з них або обома, для чого йому потрібно запустити ту саму прикладну програму.

2.5.3. Програмні компоненти розподілених систем

У розподілених системах функції одного рівня прикладної програми можуть бути рознесені між декількома комп’ютерами. Натомість програмне забезпечення, встановлене на одному комп’ютері, може відповідати за виконання функцій, що належать до різних рівнів, тому підхід до визначення розподіленої системи, за яким розподілена система – це сукупність комп’ютерів, є умовним. Щоб мати можливість описувати і реалізовувати розподілені системи, було введено поняття «програмна компонента».

Програмна компонента – це одиниця програмного забезпечення, що виконується на одному комп’ютері в межах одного процесу і надає деякий набір сервісів, які використовуються через її зовнішній інтерфейс іншими компонентами, що виконуються на цьому ж комп’ютері та на віддалених комп’ютерах (рис. 2.35). Певні компоненти інтерфейсу користувача надають свій сервіс кінцевому користувачеві.

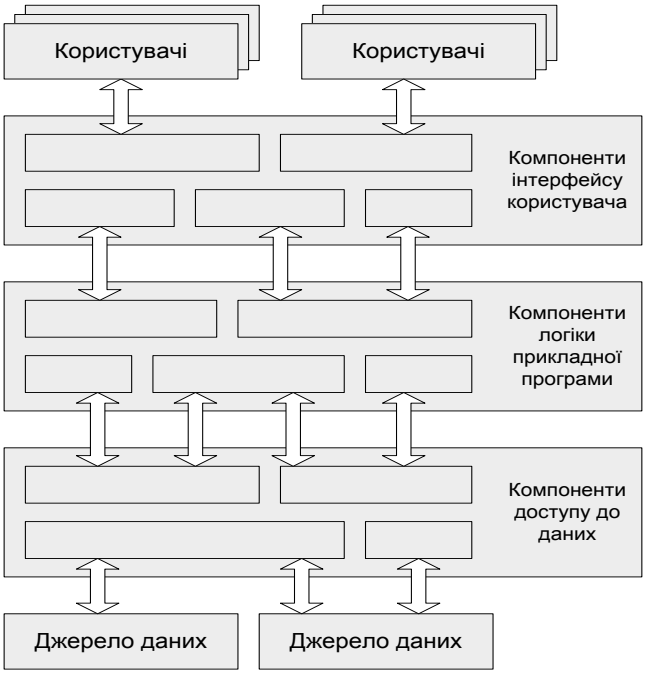


Рис. 2.35. Компоненти розподіленої системи

Виходячи з визначення програмної компоненти, можна дати більш точне визначення розподіленої системи, відповідно до якого **розподілена система** – це набір програмних компонент, які взаємодіють між собою і виконуються на одному або декількох зв'язаних комп'ютерах, становлять з погляду користувача системи єдине ціле. У такому разі прозорість є атрибутом розподіленої системи. Якщо система функціонує коректно, то від кінцевого користувача має бути приховано, де і як виконуються його запити.

Програмна компонента є мінімальною складовою розподіленої системи, що дозволяє у процесі модернізації системи одні компоненти оновлювати незалежно від інших.

У добре спроектованій системі функції кожної компоненти стосуються тільки до одного рівня прикладної програми, однак розподіл на три рівні є недостатнім для класифікації компонент. Наприклад, одна частина компонент інтерфейсу користувача може взаємодіяти з користувачем, а друга – надавати свої сервіси іншим компонентам, але з користувачем не взаємодіяти.

Такі класифікації наявні, однак вони не є загальноприйнятими й часто значною мірою залежать від специфіки конкретних прикладних програм автоматизації діяльності підприємств, організацій, компаній.

Опис програмної компоненти. Ключовим сервісом проміжного середовища розподілених систем є сервіс забезпечення обміну даними між компонентами розподіленої системи.

Щоб повністю формально описати взаємодію двох компонент розподіленої системи, необхідні три мови:

– мова повідомлень, яка описує результат серіалізації об'єктів;

– мова опису специфікацій повідомлень, що визначає коректність повідомлень для сервісів компоненти;

– мова опису інтерфейсу компоненти, яка призначена для подання набору її сервісів.

Мови опису інтерфейсу і специфікацій повідомлень часто на практиці об'єднуються та подаються однією мовою програмування.

Програмна компонента, що звертається до сервісів, для роботи з ними має повністю узгоджені власні інтерфейси з інтерфейсами сервісів. Незважаючи

на суттєві відмінності між моделлю передачі повідомлень і моделлю віддаленого виклику, для них інтерфейс компоненти розподіленої системи можна описати як сукупність адрес і форматів повідомлень її сервісів. Роль сервісу, що надається програмною компонентою, відіграє одне з таких понять:

– методи об'єкта, який активується сервером;

– об'єкт, що активується клієнтом разом зі своїми полями, властивостями й методами;

– черга з повідомленнями – запитами, які зчитуються програмною компонентою.

Адреса сервісу залежить від проміжного середовища і складається з мережної адреси компоненти і певного публічного імені сервісу. Мережна адреса програмної компоненти залежить від імені її комп'ютера для систем віддаленого виклику або адреси менеджера черги для систем обміну повідомленнями. Така адреса є адресою протоколу нижнього рівня, на якому ґрунтується певне проміжне середовище. Роль такого протоколу можуть виконувати протоколи HTTP, TCP, NetBIOS або інший протокол нижнього рівня проміжного середовища. Складовою адреси сервісу також є його ідентифікатор, роль якого може відігравати певний ідентифікатор класу, який активується для середовищ віддаленого виклику або ж ім'я черги повідомлень, з якої сервіс зчитує повідомлення запиту. Хоча ім'я викликаного методу часто зазначено в самому повідомленні, його варто розглядати як складову частину адреси сервісу, оскільки формати повідомлень неоднакові для різних методів того самого класу.

Якщо компонента системи передачі повідомлень надсилає повідомлення - відповіді клієнтові, то можна вважати, що сервіс такої компоненти має дві адреси – одну для черги запитів і другу для черги відповідей (ім'я черги відповідей може бути задано й у повідомленні запиту).

Крім інформації про повну адресу сервісу, програмі - клієнтові компоненти необхідно знати формат повідомлень, які одержуються і повертаються сервісу. Повідомлення, які одержуються, – повідомлення з параметрами віддаленого виклику і повідомлення - запити в чергах повідомлень, а повідомлення, які повертаються, є повідомлення з результатом виконання методу і повідомлення - відповіді. До параметрів методу віддаленого виклику варто

віднести й деякий ідентифікатор активованого об’єкта сервера у разі активації об’єктів за запитом клієнта. Можна стверджувати, що кожному сервісу компоненти мають відповідати єдина специфікація формату одержаних ним повідомлень і єдина специфікація прийнятих від нього повідомлень (інколи ця специфікація інформує про те, що немає відповіді від компоненти).

Важливою відмінністю систем обміну повідомленнями від систем віддаленого виклику є те, що немає обмежень на формат повідомлення. Таким чином, формально в них є можливість використовувати для опису формат повідомлення, наприклад, контекстно вільних формальних граматик. Однак було б природно вважати, що формат повідомлення має бути еквівалентним опису полів деякого класу CLI (Call Level Interface), об’єкт якого перетвориться в результаті серіалізації в передане повідомлення.

Якщо кожне повідомлення в системах черг повідомлень і параметри методу віддаленого виклику становитимуть єдиний серіалізований об’єкт деякого складного типу даних, то відмінності між системами з активованими сервером об’єктами й системами передачі повідомлень стають мінімальними. Крім того, єдиний параметр віддаленого виклику є вирішенням проблеми недоступності у повідомленні властивостей активованих сервером об’єктів, тому рекомендують створювати віддалені методи з єдиним параметром складного типу.

Таким чином, кожний сервіс програмної компоненти характеризують за трьома складовими: повною адресою сервісу; єдиною специфікацією одержаних сервісом повідомлень (запитів); єдиною специфікацією прийнятих від сервісу повідомлень (відповідей).

Сукупність специфікацій усіх сервісів програмної компоненти утворює її **інтерфейс** (рис. 2.36).

Рис. 2.36. Інтерфейс компоненти розподіленої системи

Оскільки повідомлення є результатом серіалізації певного класу, то однією зі специфікацій повідомлення можна вважати сукупність серіалізованих полів і властивостей об'єкта, маршализованого за значенням. Для систем віддаленого виклику специфікацією інтерфейсу може бути, наприклад, опис класу .NET. Таким чином, метадані зі списків з описом інтерфейсу або класу віддаленого об'єкта і класами параметрів його методів повністю визначають інтерфейс програмної компоненти. Однак такий підхід часто незручний, оскільки, хоч і зумовлює відкритість системи, але й ставить у залежність опис інтерфейсу програмної компоненти від засобу розробки, використовуваного для її створення, та вимагає надання клієнтові списків із класами компоненти. У зв'язку з цим існує потреба в загальноприйнятих і незалежних від засобів розробки програмних компонент мовах опису їх інтерфейсу.

2.5.4. Основи мережної взаємодії

З точки зору одного з комп'ютерів розподіленої системи, всі інші машини, які належать до неї, є віддаленими обчислювальними системами. Теоретичною основою мережної взаємодії віддалених систем є загальновідома еталонна модель взаємодії відкритих систем (Open Systems Interconnection Reference Model, OSI), яку розроблено Міжнародною організацією стандартизації (International Standards Organization, ISO) та названо OSI/ISO (Open Systems Interface, OSI), що поділяє процес взаємодії двох сторін на сім рівнів: фізичний, канальний, мережний, транспортний, сеансовий, подання, прикладний (рис. 2.37).

У мережах найпоширенішого стека протоколів TCP/IP протокол TCP є протоколом транспортного рівня, а протокол IP – протоколом мережного рівня. Нині забезпечує інтерфейс до транспортного рівня мережна компонента операційної системи, надаючи інтерфейс для верхніх рівнів, який використовує **сокети**, які у свою чергу, забезпечують примітиви низького рівня для безпосереднього обміну потоком байт між двома процесами. Стандартного рівня

представлення або сеансового рівня у стеці протоколів TCP/IP немає, іноді до них відносять захищені протоколи SSL/TLS.

Використання протоколу TCP/IP за допомогою сокетів надає стандартний, міжплатформний, але низькорівневий сервіс для обміну даними між компонентами. Щоб задовольнити вимоги до розподілених систем функції сеансового рівня й рівня представлення має виконувати проміжне програмне забезпечення (рис. 2.37), яке гарантуватиме відкритість, масштабованість і стійкість розподілених систем. Щоб досягнути цієї мети, проміжне середовище має надавати сервіси для взаємодії компонент розподіленої системи, зокрема такі:

– забезпечення єдиного й незалежного від операційної системи механізму використання одними програмними компонентами сервісів інших компонент;

– забезпечення безпеки розподіленої системи, тобто аутентифікація й авторизація всіх користувачів сервісів компоненти й захист переданої між компонентами інформації від перекручування й читання третіми сторонами;

– забезпечення цілісності даних або керування транзакціями, розподіленими між віддаленими компонентами системи;

– балансування навантаження на сервери із програмними компонентами;

– виявлення віддалених компонентів.

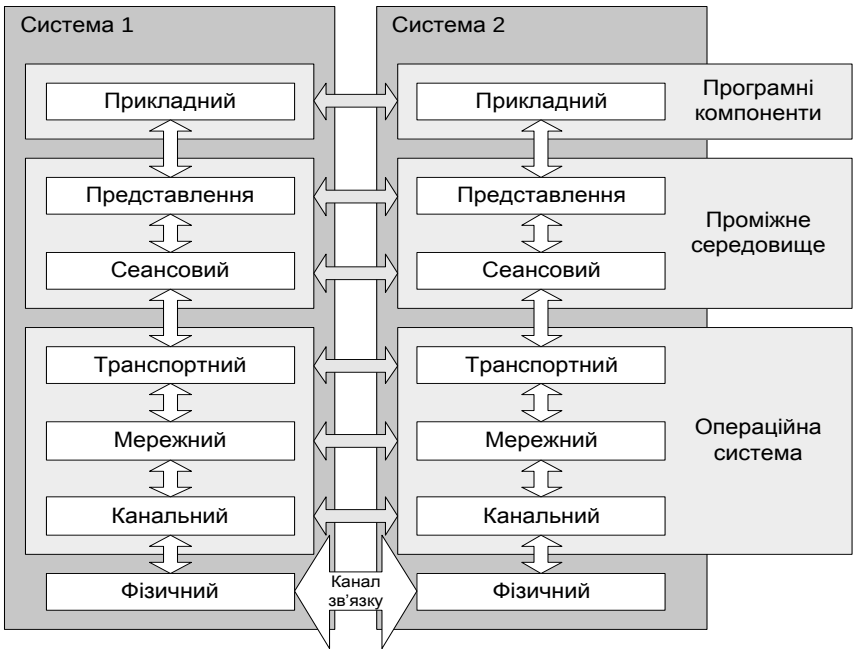


Рис. 2.37. Модель взаємодії обчислювальних систем

У межах однієї розподіленої системи може використовуватися кілька видів проміжних середовищ (рис. 2.38). Якщо підхід до проектування системи правильний, то її кожна розподілена компонента надає свої сервіси, які належать до єдиного проміжного середовища, а також використовує сервіси інших компонент, які також належать до певного проміжного середовища, однак ці середовища можуть бути різними.

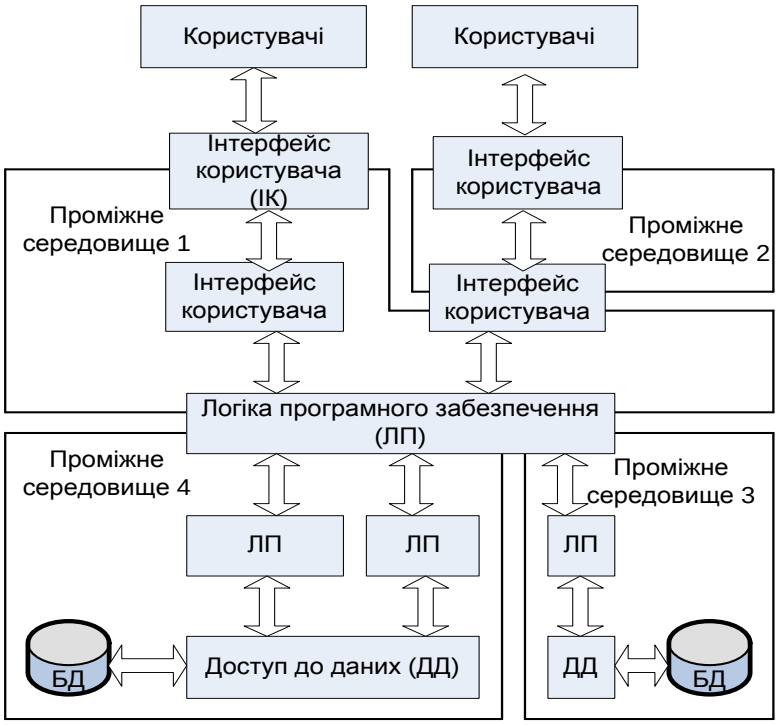


Рис. 2.38. Гетерогенна розподілена система

Розподілену систему, компоненти якої використовують кілька проміжних середовищ, можна називати гетерогенною, на відміну від гомогеної, яка ґрунтується на єдиному проміжному середовищі. Оскільки проміжне середовище може бути реалізовано на різних апаратних платформах і операційних системах, то обидва класи розподілених систем можуть мати комп’ютери, керовані як однією, так і різними операційними системами.

Нині немає проміжного середовища універсального застосування, хоча є певний рух у цьому напрямку. Основною причиною відсутності такого середовища є частково суперечливі вимоги до розподілених систем, а також різний характер мережних з’єднань між компонентами системи: наприклад, взаємодія

компонент усередині одного підприємства, ймовірно, може бути побудована інакше, ніж взаємодія компонент двох різних підприємств, що не повністю довіряють один одному.

Програмні компоненти у межах одного комп'ютера також взаємодіють за допомогою проміжного середовища, але у разі використання декількох проміжних середовищ така взаємодія може бути незручною і неефективною. В ідеальному випадку розподілену компоненту слід реалізувати так, щоб перехід від одного проміжного середовища до іншого відбувався за рахунок змінювання конфігурації програмної компоненти, а не вихідного коду. На практиці таку вимогу реалізувати непросто, однак необхідно мінімізувати можливі корегування програмного коду в разі зміни проміжного середовища.

2.6. Взаємодія компонент розподіленої системи

2.6.1. Концепції взаємодії компонент розподіленої системи

Нині розрізняють дві концепції взаємодії програмних компонент: обмін повідомленнями між компонентами й виклик процедур або методів об'єкта віддаленої компоненти за аналогією до локального виклику процедури. Оскільки будь-яка взаємодія між віддаленими компонентами ґрунтується на сокетах TCP/IP, базовим з погляду проміжного середовища є низькорівневий обмін повідомленнями на основі мережних сокетів, сервіс яких ніяк не визначає формату переданого повідомлення. На основі протоколів TCP або HTTP потім можна побудувати прикладні протоколи обміну повідомленнями більш високого рівня абстрагування, щоб реалізувати складніший обмін повідомленнями або віддалений виклик процедур.

Віддалений виклик є моделлю, що походить від мов програмування високого рівня, а не від реалізації інтерфейсу транспортного рівня мережних протоколів, тому протоколи віддаленого виклику мають обов'язково ґрунтуватися на будь-якій системі передачі повідомлень, включаючи як безпосереднє використання сокетів TCP/IP, так інші проміжні середовища для обміну повідомленнями. Під час реалізації високорівневих служб обміну повідомленнями, у свою чергу, можуть використовуватися віддалений виклик процедур, в основі яко-

го лежить більш низькорівнева передача повідомлень, що використовує, наприклад, безпосередньо мережні сокети. Таким чином, одне проміжне середовище може використовувати для свого функціонування сервіси іншого, аналогічно до того, як один протокол транспортного або мережного рівня може працювати поверх іншого у разі тунелювання протоколів.

2.6.2. Обмін повідомленнями

Розрізнюють два методи передачі повідомлень від однієї віддаленої системи до другої: безпосередній обмін повідомленнями й використання черг повідомлень. У разі безпосереднього обміну передача відбувається прямо, і можлива лише за умови, що сторона, яка приймає, готова одержати повідомлення в цей самий момент, інакше використовується посередник – менеджер черг повідомлень, тобто компонента надсилає повідомлення в одну із черг менеджера, після чого вона може продовжити свою роботу. Надалі сторона, яка отримує повідомлення, вилучить його із черги менеджера й розпочне обробку.

Найпростішим видом реалізації безпосереднього обміну повідомленнями є використання транспортного рівня мережі через інтерфейс сокетів, минаючи будь-яке проміжне програмне забезпечення. Однак такий спосіб взаємодії зазвичай не застосовують у розподілених системах, оскільки в цьому разі реалізують усі функції проміжного середовища розробники прикладних програм. За такого підходу складно отримати розширювану й надійну розподілену систему, тому для розробки прикладних розподілених систем здебільшого використовують системи черг повідомлень.

Наявні кілька розробок у сфері проміжного програмного забезпечення, що реалізують високорівневі сервіси для обміну повідомленнями між програмними компонентами, зокрема Microsoft Message Queuing, IBM MQSeries і Sun Java System Message Queue, ці системи дають можливість прикладним програмам використовувати такі базові примітиви обробки черг:

- додати повідомлення в чергу;
- взяти перше повідомлення із черги, процес блокується до появи в черзі хоча б одного повідомлення;

– перевірити чергу на наявність повідомлень;
– установити обробник, який викликається з появою повідомлень у черзі.

Менеджер черги повідомлень у таких системах може перебувати на комп’ютері, розміщеному окремо від комп'ютерів з компонентами, що беруть участь в обміні. У цьому разі повідомлення спочатку поміщується у вихідну чергу на комп’ютері з компонентою, що надсилає повідомлення, а потім пересилається менеджерові необхідної компоненти. Для створення великих систем обміну повідомленнями може використовуватися маршрутизація повідомлень, за якої повідомлення не передаються прямо менеджерові, що підтримує чергу, а проходять через низку проміжних менеджерів черг повідомлень (рис. 2.39).

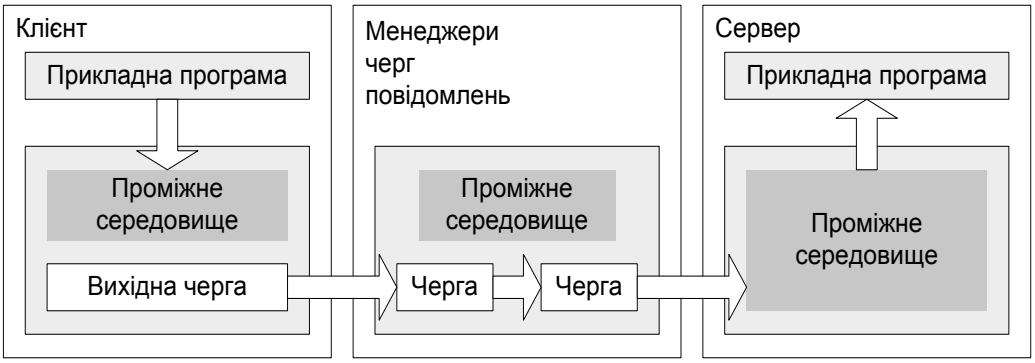


Рис. 2.39. Системи черг повідомлень

Використання черг повідомлень орієнтовано на асинхронний обмін даними. Основні переваги таких систем:

– час функціонування сервера може не залежати від часу роботи клієнтів;
– незалежність проміжного середовища від засобу розробки компонент і мови програмування;
– зчитувати й обробляти заявки із черги можуть кілька незалежних компонент, що дає можливість досить просто створювати стійкі й масштабовані системи.

Недоліки систем черг повідомлень такі:

– необхідність явного використання черг розподіленою прикладною програмою;

– складність реалізації синхронного обміну;

– певні витрати на використання менеджерів черг;

– складність отримання відповіді, оскільки передача відповіді може вимагати окремої черги на кожний компонент, що надсилає заявки.

2.6.3. Віддалений виклик процедур

Ідея **віддаленого виклику процедур** (Remote Procedure Call, RPC) з'явилася в середині 80-х років і полягала в тому, що за допомогою проміжного програмного забезпечення функцію на віддаленому комп'ютері можна викликати так само, як і функцію на локальному комп'ютері. Щоб віддалений виклик відбувався прозоро з погляду прикладної програми, яка виконує виклик, проміжне середовище має надати **процедуру-заглушку** (stub), що буде викликатися клієнтською прикладною програмою. Після виклику процедури - заглушки проміжне середовище надає переданим їй аргументам виду, придатного для передачі за транспортним протоколом, і спрямовує їх на віддалений комп'ютер з викликуваною функцією. На віддаленому комп'ютері параметри вилучаються проміжним середовищем з повідомлення транспортного рівня й передаються викликуваній функції (рис. 2.40). Аналогічно на клієнтську машину надсилається результат виконання функції, викликаної з сервера.

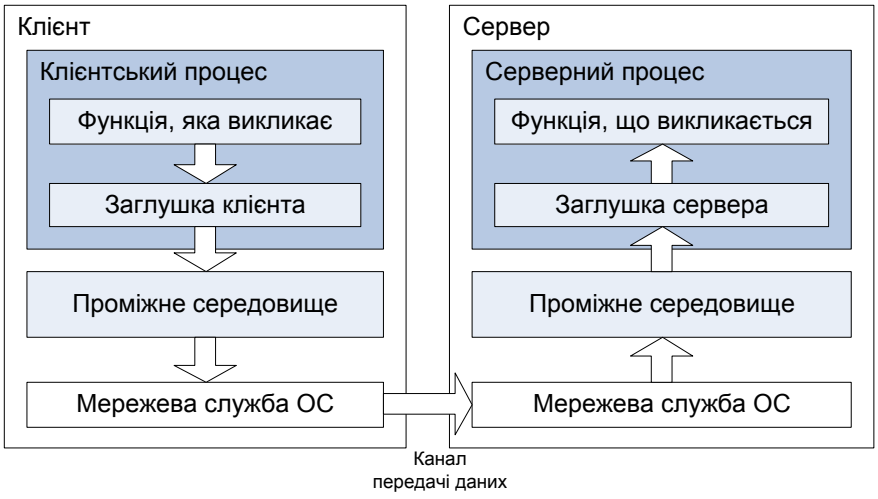


Рис. 2.40. Віддалений виклик процедур

Розрізняють три можливі варіанти віддаленого виклику процедур:

Синхронний виклик – клієнт очікує завершення процедури сервером і, у разі потреби, отримує від нього результат виконання віддаленої функції.

Однонаправлений асинхронний виклик – клієнт продовжує виконувати своє завдання, не отримуючи відповіді від сервера, відповідь або відсутня, або її реалізація якось інакше передбачена під час розробки (наприклад, через функцію клієнта, яку віддалено викликає сервер).

Асинхронний виклик – клієнт продовжує виконувати завдання, після завершення сервером процедури він отримує повідомлення й результат її виконання, наприклад через callback-функцію, яка викликається проміжним середовищем під час одержання результату від сервера.

Процес перетворення параметрів для їх передачі між процесами (або доменами прикладної програми у разі використання .NET) під час віддаленого виклику називають **маршалізацією** (marshalling). Перетворення екземпляра якого-небудь типу даних у придатний для передачі за межі викликаючого процесу набір байтів називаються **серіалізацією.**

Десеріалізація – процедура, обернена серіалізації, – полягає у створенні копії серіалізованого об’єкта на основі отриманого набору байтів. Такий підхід до передачі об’єкта між процесами за допомогою створення його копій називають **маршалізацією за значенням** (marshal by value), на відміну від **маршалізації за посиланням**.

Маршалізація за посиланням під час передачі параметрів за посиланням використовує серіалізацію не самих вказівників, а об’єктів, на які вказують вказівники.

Процес серіалізації повинен бути визначений для всіх типів даних, переданих у процесі віддаленого виклику, зокрема для параметрів функції, яка викликається й результату, який повертає функція. У разі передачі параметрів за посиланням сериалізації підлягають об’єкти, на які посилаються, до самих вказівників сериалізація не може бути застосована, оскільки це ускладнює використання механізму віддаленого виклику в мовах, які підтримують вказівники на об’єкти невідомого типу.

2.6.4. Використання віддалених об’єктів

У зв’язку з переходом розробників прикладних програм від структурної парадигми до об’єктної з’явилася необхідність у використанні віддалених об’єктів (remote method invocation, RMI). Віддалений об’єкт являє собою деякі дані, сукупність яких визначає його стан, який можна змінювати за рахунок виклику його методів. Зазвичай можливий прямий доступ до даних віддаленого об’єкта, за якого виконується неявний віддалений виклик, необхідний для передачі значення поля даних об’єкта між процесами. Методи й поля об’єкта, які можна використовувати через віддалені виклики, доступні через деякий зовнішній інтерфейс класу об’єкта. Зовнішній інтерфейс компоненти розподіленої системи в таких системах зазвичай збігається із зовнішнім інтерфейсом одного з класів, на якому побудована компонента.

У момент, коли клієнт починає використовувати віддалений об’єкт, з боку клієнта створюється клієнтська заглушка, яку називають **посередником** (proxy), який реалізує той самий інтерфейс, що й віддалений об’єкт. Процес, який виконує виклик, використовує методи посередника, який маршалізує їх параметри для передачі мережею, і передає їх мережею серверу. Проміжне середовище, яке десеріалізує параметри й передає їх заглушці з боку сервера, називають **каркасом** (skeleton) або, як і у разі віддаленого виклику процедур, **заглушкою** (рис. 2.41). Каркас зв’язується з деяким екземпляром віддаленого об’єкта, як заново створеним, так і наявним екземпляром об’єкта, залежно від застосовуваної моделі використання віддалених об’єктів.

Весь описаний процес називають **маршалізацією віддаленого об’єкта за посиланням** (marshal by reference). На відміну від маршалізації за значенням, екземпляр об’єкта перебуває у процесі сервера й не залишає його, а для доступу до об’єкта клієнти використовують посередників. Уразі маршалізації за значенням саме значення об’єкта серіалізується в набір байтів для його передачі між процесами, після чого необхідне створення його копії в іншому процесі.

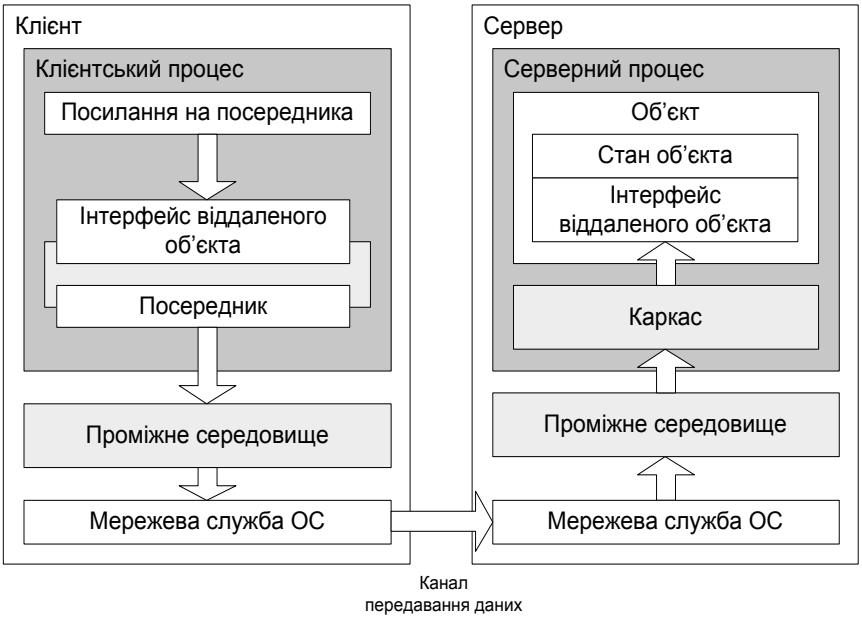


Рис. 2.41. Використання віддалених об'єктів

Аналогічно до віддаленого виклику процедур, виклик методу віддаленого об'єкта може бути як синхронним, так і асинхронним. Процесу використання віддалених об'єктів, що не зустрічалися у віддаленому виклику процедур, характерні такі особливості. По-перше, якщо на момент формування концепції віддаленого виклику процедур виключення (exceptions) ще не підтримувалися й не використовувалися найбільш поширеними мовами програмування, то надалі вони стали методом інформування сторони, яка виконує виклик, про проблеми, що виникли у сторони, яка викликається. Таким чином, у системах, що використовують віддалені об'єкти, серіалізації підлягають як параметри методу і його результат, так і виключення, які з'являються у процесі виконання віддаленого методу. По-друге, параметр або результат методів можуть теж передаватися посилання на віддалений об'єкт (рис. 2.42), якщо віддалений метод – клієнт, який виконує виклик, також є сервером RMI.

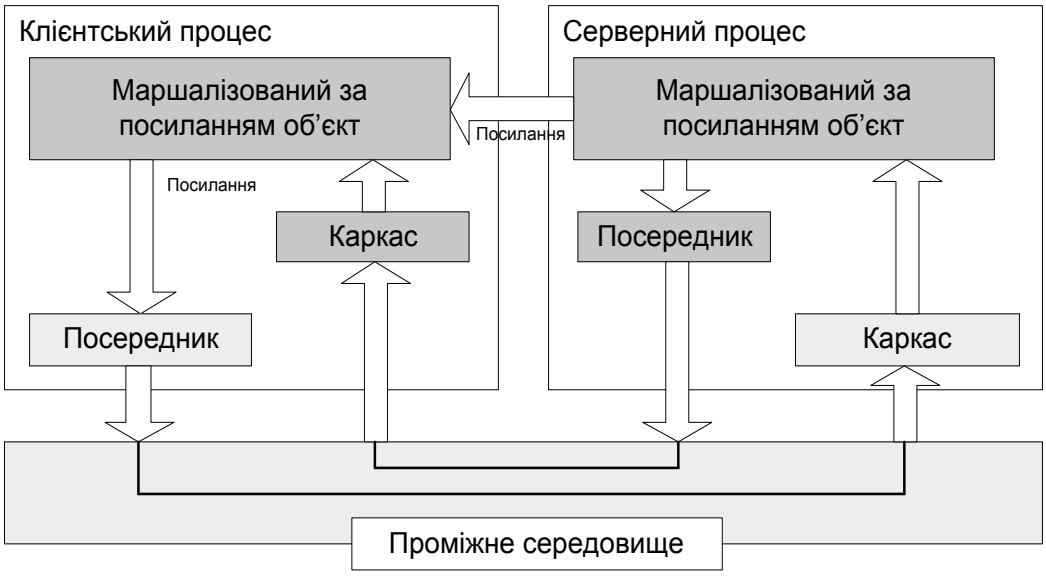


Рис. 2.42. Передача віддаленому методу посилання на об'єкт, що маршалізується за посиланням

У разі використання віддалених об'єктів важливими є питання про час їх функціонування: у який момент часу утворюється екземпляр віддаленого об'єкта; протягом якого проміжку часу він існує.

Щоб описати життєвий цикл у системах із віддаленими об'єктами, використовують такі поняття прикладного програмування:

– **активацію об'єкта** – процес переведення створеного об'єкта у стан обслуговування віддаленого виклику, тобто зв'язування його з каркасом і посередником;

– **деактивацію об'єкта – процес переведення об'єкта у стан невикористання.**

Розрізняють три моделі використання віддалених об'єктів: модель єдиного виклику (singlecall), модель єдиного екземпляра (singleton), а також модель активації об'єктів за запитом клієнта (client activation). Перші дві моделі також іноді називають моделями серверної активації (server activation), хоча активація завжди відбувається на сервері після будь-якого запиту від клієнта.

Модель єдиного виклику. В разі використання цієї моделі об'єкт активується на час єдиного віддаленого виклику. В найпростішому випадку для кожного виклику віддаленого методу об'єкта клієнтом на сервері створюється

й активується новий екземпляр об’єкта, що деактивується й потім знищується відразу після завершення віддаленого виклику методу об’єкта. Таким чином, віддалені виклики різних клієнтів ізольовано один від одного. За рахунок видалення об’єктів після виклику досягається раціональна витрата ресурсів пам’яті, але можуть витрачатися значні ресурси процесора на постійне створення об’єктів. Програма-посередник на клієнті й заглушка на сервері існують до знищення посередника об’єкта (рис. 2.43).

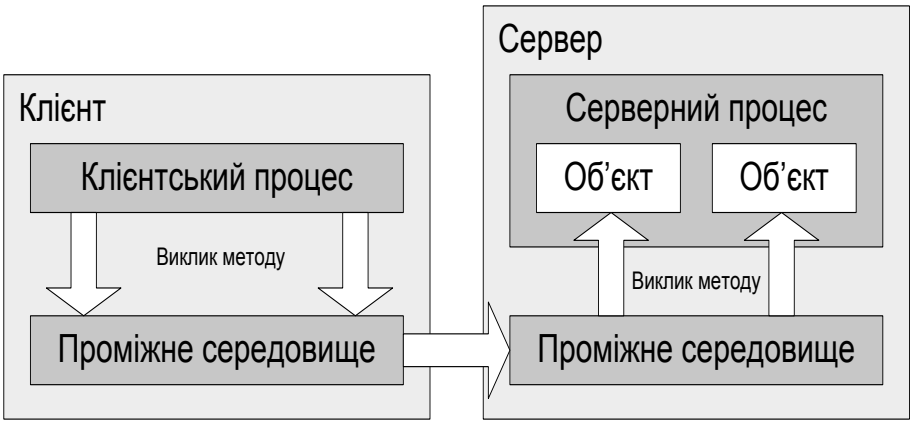


Рис. 2.43. Режим єдиного виклику віддаленого методу

Такий метод застосування віддалених об’єктів можна розглядати як деякий варіант віддаленого виклику процедур, оскільки об’єкт не зберігає свого стану між викликами. Проте сервер використовує свої ресурси для підтримки каркасу й каналу між посередником і заглушкою.

Недоліком методу одного виклику є часте створення й видалення екземплярів об’єктів, тому в проміжному середовищі може міститися сервіс, що дозволяє підтримувати деяку кількість уже створених, але ще не активованих об’єктів, які використовуються для обробки віддалених викликів. Такий набір об’єктів, що очікують своєї активації, називають **пулом об’єктів** (object pooling). Після завершення віддаленого виклику об’єкти деактивуються й можуть бути поміщені в пул і використані повторно надалі або видаляються, якщо розмір пулу досягнув деякого максимального значення. Така технологія дозволяє отримати баланс між швидкістю обробки запиту й обсягом використовуваних ресурсів сервера. Як бачимо з опису, в системі з пулом об’єктів

активація не завжди потрібна безпосередньо після створення об’єкта, а видалення не завжди виконується відразу після деактивації.

Відмітною рисою методу одного виклику є мінімальні витрати на організацію системи балансування навантаження та її найбільша ефективність, оскільки кожний сервер, який обслуговує запити, може обробити виклик будь-якого віддаленого методу.

Модель єдиного екземпляра. У разі використання моделі єдиного екземпляра віддалений об’єкт існує не більш ніж в одному екземплярі. Створений об’єкт існує, поки є хоч один клієнт, який його використовує (рис. 2.44).

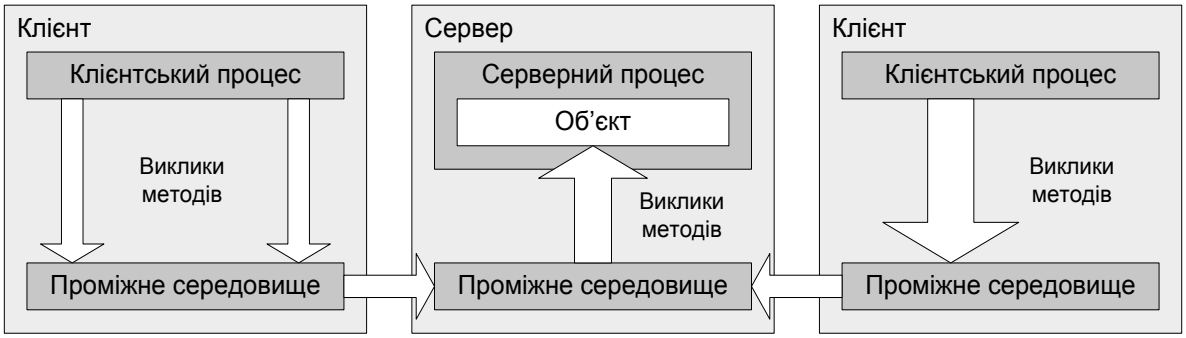


Рис. 2.44. Використання віддалених об’єктів у режимі єдиного екземпляра

У разі використання моделі єдиного об’єкта виклики різних клієнтів працюють одним екземпляром віддаленого об’єкта. Оскільки виклики клієнтів не ізольовані один від одного, то використовуваний об’єкт не повинен мати будь-якого внутрішнього стану. Модель єдиного об’єкта дозволяє отримати найбільш високу продуктивність, оскільки об’єкти не створюються й не активуються сервером під час кожного виклику методу об’єкта.

Активація за запитом клієнта. Під час кожного створення клієнтом посилання на віддалений об’єкт (точніше, на посередника) на сервері виникає новий об’єкт, що існує, поки клієнт не видалить посилання на посередника. За такого методу виклики різних клієнтів ізольовано один від одного і кожний об’єкт зберігає свій стан між викликами, що призводить до найменш раціонального використання ресурсів пам’яті сервера (рис. 2.45).

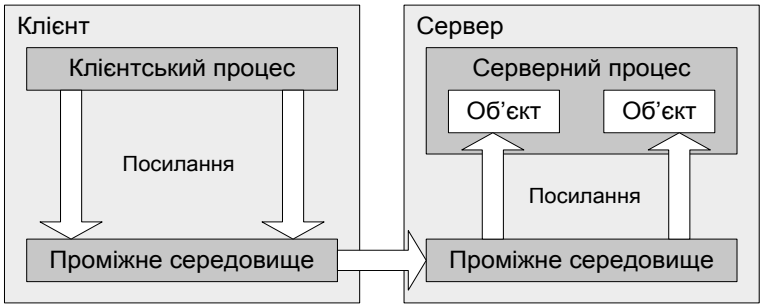


Рис. 2.45. Об'єкти, що активуються клієнтом

Стан компоненти розподіленої системи. Програмні компоненти з погляду користувачів сервісів можна поділити на дві категорії: компоненти без внутрішнього стану**, що зберігається між віддаленими викликами своїх методів (***stateless components***);** компоненти із внутрішнім станом**, що зберігається між віддаленими викликами своїх методів (***statefull components***).**

Під станом у цьому разі розуміють сукупність значень полів об'єктів, які реалізують компоненти, що зберігаються в пам'яті сервера. Якщо компонента у процесі своєї роботи зберігає які-небудь дані в зовнішньому сховищі, наприклад у базі даних або у черзі повідомлень, це зазвичай не розглядається як її внутрішній стан.

Модель єдиного виклику не зберігає стану віддаленого об'єкта між викликами його методів, тому таку модель можна використовувати лише з розподіленими компонентами без внутрішнього стану. Модель одного екземпляра може бути використана для виклику компонентів із внутрішнім станом, але це нераціонально, оскільки її стан буде змінюватися кожним з клієнтів у довільному порядку. Модель активації за запитом клієнта може застосовуватися з будь-якими компонентами, але для компонент без внутрішнього стану такий підхід зазвичай зумовлює непродуктивне використання пам'яті за деякого виграшу у витратах часу процесора порівнянно з моделлю одного виклику.

Компоненти без збереження внутрішнього стану, які використовують разом з моделлю єдиного виклику з пулом об'єктів, мають найбільші можливості масштабування системи за оптимального балансу між витратами пам'яті й навантаженням на процесор.

2.6.5. Розподілені події

У процесі розробки програмного забезпечення часто потрібно отримувати повідомлення про які-небудь події, що виникають асинхронно, тобто в деякі довільні моменти. У розподілених системах також є необхідність використання таких повідомлень, що отримуються від віддаленої системи. Можна визначити два підходи до обробки подій: **тісно пов'язані й слабопов'язані події**.

У разі тісно пов'язаної події виконується пряме повідомлення однієї сторони іншою стороною. Незважаючи на те, що цей метод можна використовувати разом з однонаправленим асинхронним викликом, йому властива низка недоліків, що обмежують його застосування в розподілених системах, зокрема:

– обидві компоненти системи мають виконуватися одночасно;

– для повідомлення декількох компонент про одну подію стороною, що повідомляє, мають використовуватися механізми для ведення списку одержувачів подій;

– утруднена фільтрація або протоколювання подій.

Виходячи з цього, в розподілених системах також застосовуються слабо пов'язні події, коли джерела події (видавці) не взаємодіють прямо з одержувачами подій (передплатниками). Проміжне середовище в цьому разі має надати сервіс, що дозволяє передплатникові замовити будь-яку подію або відмовитися від підписки, а видавцеві – ініціювати подію для розсилання передплатникам (рис. 2.46).

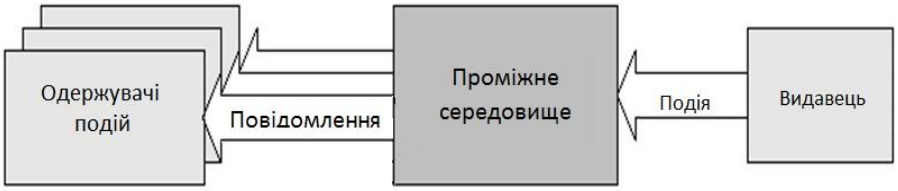


Рис. 2.46. Передплатники й видавці слабопов'язаних подій

У разі використання слабо пов'язних подій передплатники, видавці й менеджер подій можуть розташовуватися на різних комп'ютерах. Сама подія

може бути реалізована як, наприклад, виклик менеджером подій деякого зареєстрованого методу віддаленого об’єкта.

2.6.6. Розподілені транзакції

Транзакція – послідовність операцій з якими-небудь даними, що або успішно виконується повністю, або не виконується взагалі. Якщо неможливо успішно виконати всі дії, то відбувається повернення до початкових значень усіх змінених протягом транзакції даних (відкат транзакції). Транзакція має задовольняти таким вимогам:

– атомарність, тобто транзакція виконується за принципом «все або нічого»;

– погодженість – після успішного завершення або відкату транзакції всі дані перебувають у погодженому стані, їх логічна цілісність не порушена;

– ізоляція – для об’єктів поза транзакцією не видно проміжних станів, яких можуть набувати дані, що змінюються у транзакції; «зовнішні» об’єкти до успішного завершення транзакції повинні мати той самий стан, у якому перебували до її початку;

– сталість – якщо транзакція успішна, то внесені зміни повинні мати постійний характер (тобто збережені в енергонезалежній пам’яті).

Принцип роботи розподіленої транзакції показано на рис. 2.47. Транзакції є основою прикладних програм, які працюють із базами даних, однак у розподіленій системі недостатньо використовувати лише транзакції систем керування базами даних. Наприклад, у розподіленій системі у виконанні транзакції може брати участь кілька розподілених компонент, що працюють із декількома незалежними базами даних (рис. 2.47).

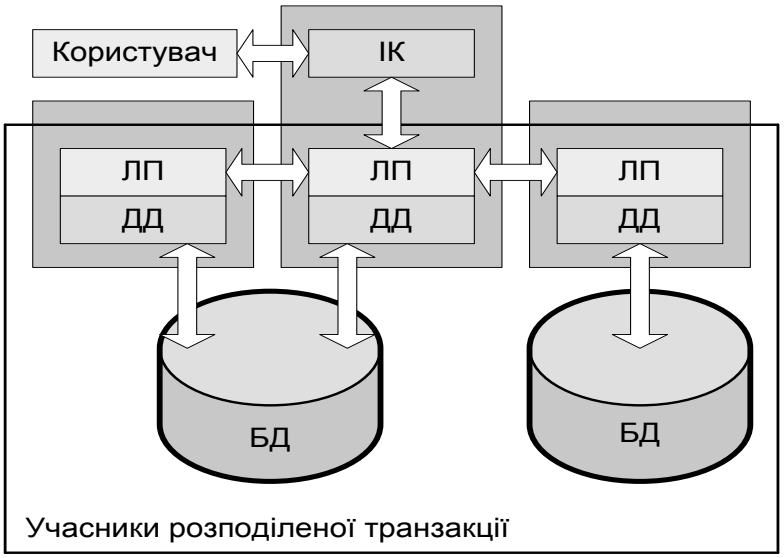


Рис. 2.47. Розподілена транзакція, де ІК – інтерфейс користувача, ЛП- логіка прикладного програмного забезпечення, ДД – доступ до даних, БД - база даних

Розподіленою називають транзакцію, яка охоплює операції декількох компонент розподіленої системи, що взаємодіють між собою, і кожна з яких може працювати з певними СУБД або іншими службами, наприклад використовувати черги повідомлень або навіть працювати з файлами. У разі відкату транзакції всі ці операції має бути скасовано, для чого необхідно виконання таких умов: проміжне середовище має підтримувати керування розподіленими між декількома компонентами транзакціями; компоненти розподіленої системи не можуть працювати з якими-небудь службами або ресурсами, які не беруть участі у транзакції.

Розподілені транзакції є найважливішим елементом підтримування цілісності даних у розподіленій системі, тому для ширшого їх застосування проміжне середовище може містити механізми, які у разі потреби (і певних витрат часу на написання коду) дозволять використовувати в розподілених транзакціях зовнішні служби, що не підтримують транзакції. Такий механізм називають компенсуючим ресурс менеджером (compensating resource manager). Компенсація в цьому разі означає повернення ресурсу до початкового стану під час відкату транзакції.

Одночасно відбувається формування і стандартизація ще одного поняття, пов'язаного з підтримкою цілісності даних у розподілених системах, – господарської діяльності (business activity). Business activity зазвичай є відображенням деякого реального процесу, наприклад купівлі в магазині, процеси якої описують від оформлення замовлення до підтвердження доставки кур'єром. Business activity може охоплювати транзакції, які включають усі процеси у точній відповідності, як вони відбуваються у реальному житті (оформлення замовлення покупця, замовлення товару в постачальника і так далі – до підтвердження покупцем доставки). На відміну від транзакції, час життя якої передбачено коротким, business activity може тривати довго (наприклад, місяць), вона може підтримувати скасування внесених змін (зокрема, оформлення повернення товару постачальнику в разі відмови покупця) за рахунок використання компесуючих завдань.

2.6.7. Безпека в розподілених системах

Для гарантування безпеки розподіленої системи проміжне середовище має забезпечувати функціонування трьох загальновідомих функцій, необхідних для створення безпечних систем:

1. Перевірка того, що користувач сервісів компоненти розподіленої системи дійсно є користувачем, якому дозволено мати доступ до сервісів та даних системи (аутентифікація http://www.intuit.ru/department/se/msfdev/2/footnote.5.1.htm). Така перевірка може бути однобічною, коли тільки сервер переконується в тому, що клієнту дозволено мати доступ, або двобічною, коли клієнт теж переконується в тому, що цей сервер надає саме ті сервіси, які йому потрібні.
2. Обмеження доступу до сервісів компонента залежно від результатів аутентифікації (авторизація). Для вирішення такого завдання проміжне середовище має підтримувати обмеження доступу, в основі якого покладено **ролі** (role based security). Оскільки немає можливості позначити рівні доступу через конкретних користувачів або груп користувачів системи, то слід використовувати деякі абстрактні ролі, які у процесі розгортання компоненти пов'язуються адміністратором системи з обліковими записами користувачів системи.

3. Захист даних, переданих між компонентами системи, від перегляду і змінювання третіми сторонами потребує, щоб передані між компонентами повідомлення візувати електронним підписом і шифрувалися як клієнтом, так і сервером.

Функції безпеки можна забезпечувати транспортним протоколом, який використовується проміжним середовищем, самим середовищем, або ними обома одночасно.

2.7. Опис інтерфейсу програмної компоненти

2.7.1. Мова і схеми XML (Extensible Markup Language)

Мова XML нині набула низки різноманітних застосувань і є основою для великої кількості загальноприйнятих специфікацій, які використовують у розподілених системах (зокрема у мовах XML, XSD, SOAP, WSDL), та основні з яких подано на рис. 2.48.

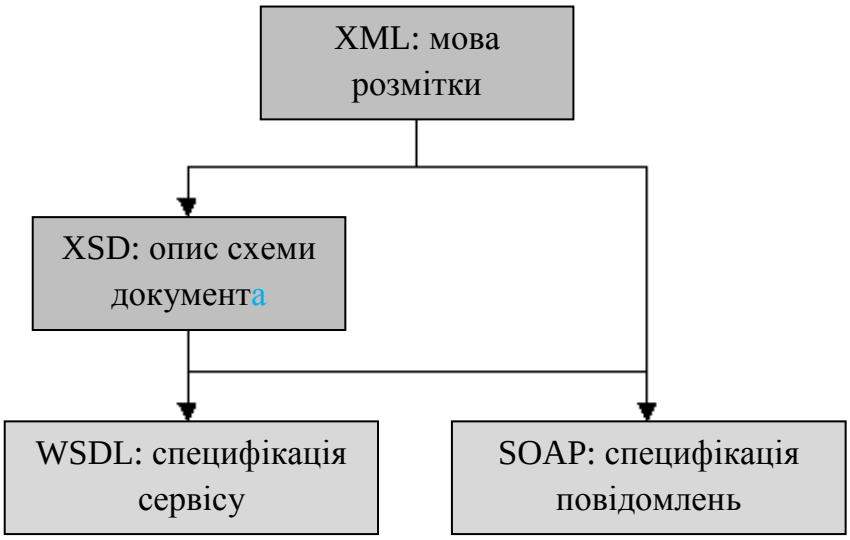


Рис. 2.48. Специфікації, які мають в основі XML-формат

Мова XML – це мова розмітки текстового документа, поданого у вигляді сукупності іменованих, деревоподібних вкладених елементів, кожний з яких може мати деяке текстове значення й набір атрибутів, у яких є ім’я і просте значення (рядок). Мова XML є абстрактною мовою розмітки, яка не визначає ніякого змісту елементів документа. Документи XML добре читає як людина, так і численні програмні аналізатори. У разі природного підходу до

імен елементів і атрибутів вона є мовою, яка самодокументується. Перед деревоподібною структурою елементів, що мають єдиний корінь, можуть перебувати окремі елементи з метаінформацією, яка свідчить, зокрема, про кодування документа й версію мови.

Основними недоліками XML з погляду обміну повідомленнями – є незручне, через його деревоподібну структуру, подання відносин «багато до багатьох», а також більші витрати часу на передачу й обробку повідомлень мовою XML порівняно з двійковим представленням аналогічних даних.

Оскільки властиве XML відкрите подання інформації не завжди зручне з погляду безпеки, то наявні специфікації XML-DigitalSignature і XML-Encrypton, призначені для передачі в XML-форматі конфіденційної інформації. Перша специфікація дозволяє додати до XML-документа цифровий підпис, друга – зашифрувати XML-документ або окремі його елементи.

Однією з переваг XML є наявність мов специфікацій, що визначають правильний XML-документ. Спочатку цю функцію виконував DTD (Document Type Definition), однак нині загальноприйнятим стандартом є специфікація схем XML (XML Schema Definition, XSD). Файл з описом схеми XML визначає такі параметри: словник документа (імена елементів і атрибутів); синтаксис коректного документа; складні типи даних.

2.7.2. SOAP: мова повідомлень розподіленої системи

Стандартизація опису мови XML дала широкі можливості для побудови на його основі мов опису повідомлень, переданих між програмними компонентами, і мов опису сервісів програмних компонент. Наприкінці 90-х років розпочали розробляти дві специфікації для побудови розподілених гетерогенних систем – SOAP і XML RPC. Специфікація XML RPC підтримується нині більшою кількістю мов, але має менше можливостей і не підтримується стандартною бібліотекою .NET Framework.

Оскільки в момент розробки таких специфікацій протокол HTTP був найпоширенішим у міжмережних екранах, то його було обрано як стандартний транспортний протокол для створення гетерогенних проміжних середовищ. Хоча специфікація SOAP не прив’язана жорстко до якого-небудь транспортного

протоколу, який використовує SOAP і WSDL, проміжне середовище названо **web-службою** (web services). Web-сервіси використовують дві мови прикладних програм – мову опису повідомлень SOAP і мову опису сервісів й інтерфейсів WSDL.

Рекомендацію SOAP спочатку розробляли як специфікацію для віддаленого виклику методів і розшифровували як Simple Object Access Protocol. Повідомлення SOAP є XML-документом, який називають **конвертом** або **пакетом** (envelope). Цей документ містить заголовки з метаінформацією в елементі soap:Header і тіло повідомлення в елементі soap:Body. У заголовках пакета міститься інформація прикладної програми, яка може використовуватися проміжним середовищем. Завдяки тому, що основний стандарт не обмежує змісту заголовків, SOAP є розширюваною специфікацією, і нині триває процес стандартизації її розширень.

Через різні причини нині розрізняють два різні способи представлення інформації в тілі пакета SOAP: кодування SOAP RPC (у двох варіантах) і кодування SOAP Document. Кодування SOAP RPC призначено винятково для передачі параметрів віддаленого виклику й визначає повідомлення як ім’я методу і список параметрів. У разі використання кодування SOAP Document, що є нині фактичним стандартом, повідомлення являє собою XML-документ, який має схему і простір імен, які задано в описі сервісу мовою WSDL (Web Service Definition Language). Зазвичай повідомлення складається з імені методу віддаленого об’єкта і списку його параметрів, сама специфікація кодування ніяк не фіксує його змісту.

Опис типів переданих даних за специфікацією SOAP Document містить схему XML, яка визначає коректні повідомлення,що отримуються програмним компонентом у тілі пакета SOAP, та включає такі параметри:

– опис вхідних і вихідних повідомлень, які зв’язуються з описаними типами даних;

– опис операцій (сервісів програмної компоненти), з кожною з яких зв’язується вхідне й вихідне повідомлення;

– опис типів портів (ідентифікаторів програмних компонент), з кожним з яких зв’язується деякий набір операцій;

– опис прив’язок (binding), що зв’язують типи портів і їх повідомлень з певним типом кодування тіла пакета, а також із версією протоколу SOAP;

– опис портів, що зв’язують типи портів і відповідні прив’язки з конкретними URL;

– загальний опис служби (інтерфейсу програмної компоненти) як сукупності портів.

2.7.3. WSDL: опис інтерфейсу програмної компоненти

Для опису інтерфейсу програмної компоненти, включаючи специфікацію коректних повідомлень, було розроблено мову WSDL. Опис мовою WSDL містить сім складових (рис. 2.49).

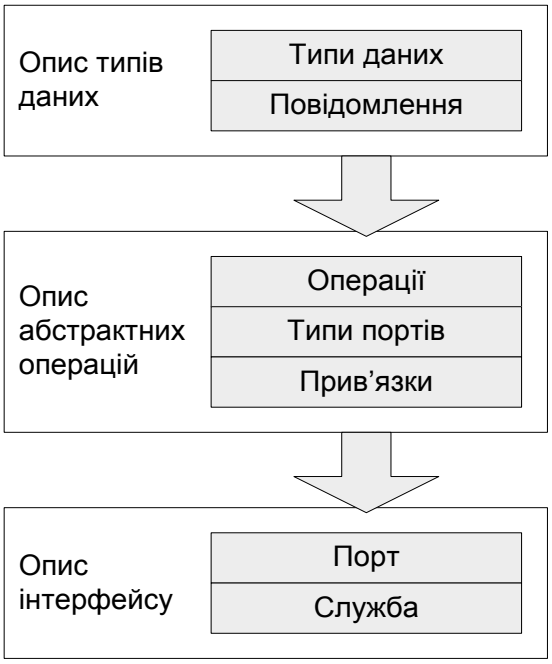


Рис. 2.49. Складові WSDL-документа

2.7.4. Серіалізація об’єктів .NET Framework

На відміну від прикладних програм на некерованому коді, прикладна програма .NET Framework не обов’язково виконується у вигляді окремих процесів, а може перебувати в межах одного процесу операційної системи у власних областях – доменах прикладної програми, які можна розглядати як деякі логічні процеси віртуальної машини Common Language Runtime (CLR).

Використання керованого коду за таких умов дозволяє гарантувати ізоляцію прикладної програми у межах своїх областей. У процесі передачі повідомлення між доменами прикладної програми деякого об’єкта для його класу слід визначати такі параметри: процедуру серіалізації, що дозволяє зберегти стан об’єкта в деякому зовнішньому сховищі (наприклад, у файлі або в повідомленні транспортного протоколу) за допомогою потоків введення-виведення; процедуру десеріалізації, що створює копію об’єкта за збереженим станом (рис. 2.50).

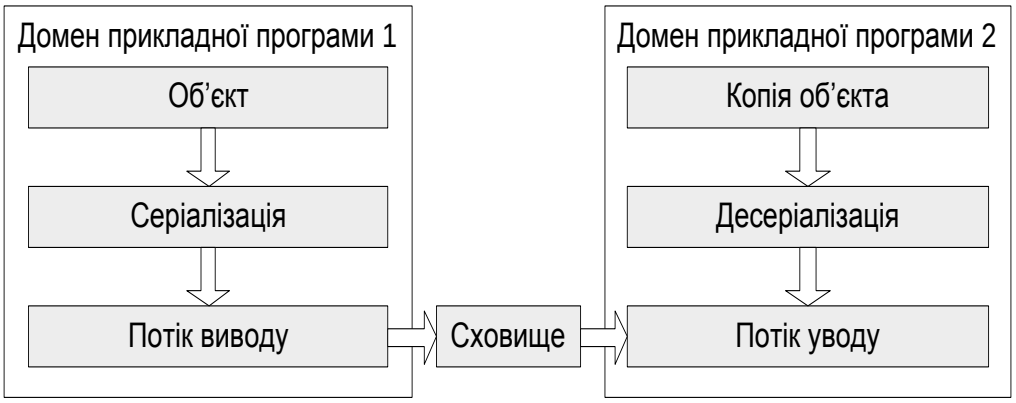


Рис. 2.50. Серіалізація й десеріалізація об’єкта

Слід зазначити, що загалом такі об’єкти можуть бути об’єктами різних класів і навіть створеними в різних системах розробки прикладних програм. Завдання серіалізації об’єкта, який містить лише поля з елементарних типів значень (спадкоємців класу *System.ValueType*) і рядків, не становить принципових труднощів. Для такого об’єкта у процесі серіалізації в потік записують самі значення всіх полів об’єкта, однак загалом об’єкт містить посилання на інші об’єкти, які, у свою чергу, можуть посилатися один на одного, утворюючи граф об’єктів (*object graph*). Самі посилання не можуть зберігатися в потоці введення-виведення, тому основне питання серіалізації – у який спосіб заміняти посилання.

Граф об’єктів – орієнтований граф $G = < V, E >$, у якому вершини – це об’єкти (множина V), а ребра направлені від об’єктів, що містять посилання, до об’єктів, на які посилаються (рис. 2.51). Усі об’єкти, які розглядаються у процесах серіалізації або десеріалізації описують у термінах об’єктно-орієнтованого підходу. В такому разі об’єкти, властивості яких наслідують інші об’єкти, є батьками, а об’єкти, які наслідують властивості батьків, є дітьми.

Наявність посилання на об’єкт *B* у об’єкті *A* є приналежністю пари <*A*,*B*> множині *E*. У графі всі поля об’єктів, які належать до простих типів значень і рядків, можна вилучити з розгляду в силу тривіальності їх серіалізації. Хоча формально рядки є посилальними типами, обмеженням реалізації рядків у CLI є те, що немає можливості змінити вже створений рядок, але є можливість тривіально обробляти рядки під час серіалізації, зберігаючи в потоці їх вміст.

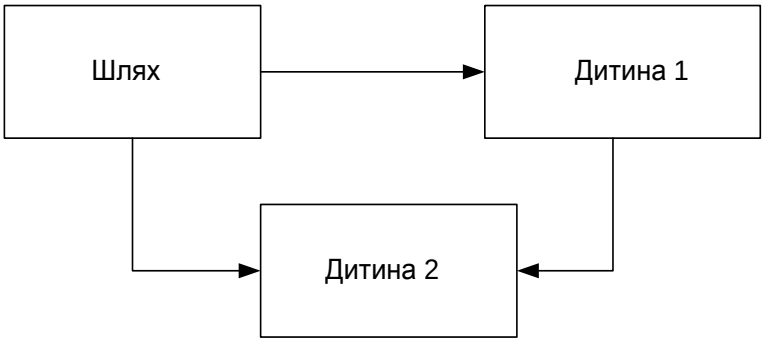


Рис. 2.51. Граф об’єктів

Існує окремий випадок, коли серіалізація виконується тривіально, у такому разі граф повинен мати вигляд орієнтованого дерева. У кожну вершину дерева, відмінну від кореня, направлене єдине ребро, існує єдина вершина, у яку не направлене жодне ребро, – корінь. За таких умов серіалізація може виконуватися від кореня методом у глибину, серіалізація полів-посилань функціонує аналогічно до серіалізації полів-значень. Якщо виявлено посилання на об’єкт, то замість нього у сховище розміщують значення полів об’єкта, на який посилаються, і деяку інформацію прикладної програми щодо типу об’єкта. Наприклад, якщо наявні ребра $<A, B_1>$, ... $<A, B_n>$, то функцію серіалізації *S* для об’єкта *A* можна подати як $S(A) = <V(A), <S(B_1), ... S(B_n)>>$, де функція *V* – значення полів-значень і рядків цього об’єкта.

Проблема серіалізації графу об’єктів полягає в тому, що посилання на той самий об’єкт може бути значенням поля різних об’єктів (наявні ребра $<A_1, B>$ і $<A_2, B>$). Можлива також наявність циклів у вигляді $A \rightarrow ... \rightarrow A$.

У цьому разі у процесі серіалізації об’єктам мають відповідати деякі ідентифікатори, й у сховищі окремо зберігається список об’єктів, позначений ідентифікаторами, а під час серіалізації замість посилань записують ідентифікатори об’єктів, на які посилаються. Якщо A_1, ... A_n – всі вершини графа, то

після серіалізації утвориться множина $\{<id_1, S(A_1)>, ... <id_{An}, S(A_n)>\}$, де $S(A) = < V(A), <id_1, ... id_{Bn}> >$, а $B_1, ... B_n$ – об'єкти, посилання на які безпосередньо міститься в об'єкті A. У програмі роль ідентифікатора об'єкта виконує його адреса, але замість неї зручніше обрати деякі ідентифікатори у процедурі серіалізації для більш легкого читання людиною отриманого образу. Під час серіалізації потрібно включити список адрес уже записаних об'єктів як для ведення списку ідентифікаторів, так і для виявлення можливих циклів у разі обходу графу методом у глибину.

Слід зазначити, що результат серіалізації дерева легко подати у вигляді XML, коли вміст кожного об'єкта є одним елементом з деяким набором атрибутів і вкладених елементів, тому, розглядаючи проблему серіалізації, можна сформулювати рекомендацію, щоб класи, передані між віддаленими компонентами, були коренем дерева об'єктів. Зокрема це дерево може бути навіть виродженим, тобто клас не містить полів-посилань взагалі.

2.8. Базові технології подання інформації в розподілених системах

2.8.1. Вимоги до прикладних програм серверної сторони

Розглядаючи платформи для створення прикладних програм серверної сторони, необхідно виокремити такі основні підходи: безпосередня обробка запитів і формування відповідей; вбудовування програмного коду в шаблони HTML-сторінок.

Перший підхід надає найбільші можливості з керування обробкою і підвищенням продуктивності, оскільки він передбачає передачу всіх даних про запит безпосередньо виконуваного коду, який може як сформувати відповідь зі сторінкою для користувача, так і відкрити процес передачі потоку бітів, наприклад для передачі зображення. Однак за такого підходу всі дані для передачі формуються програмно, що уповільнює розробку простих сторінок і ускладнює взаємодію між розробником дизайну сторінки і програмістом. Прикладами цього підходу є технології CGI (Common Gateway Interface), Java Servlets.

Другий підхід використовує шаблони сторінок користувача, оформлені таким чином, щоб дозволити вставляти в них ділянки програмного коду. Цей підхід особливо ефективний під час створення простих прикладних програм, основна інформація в яких статична, а динамічна інформація може бути генерована простими програмними конструкціями. У процесі розробки складних програмних систем цей варіант ускладнює взаємодію між компонентами і реалізацію складної архітектури, а також він менш ефективний за продуктивністю й обмежує можливості з реалізації складних сторінок. Прикладами цього підходу є найпоширеніші нині технології Personal Home Page (PHP), Active Server Pages (ASP), Java Server Pages (JSP).

Крім різних підходів до генерації сторінок сучасні платформи розробки складних Web-систем мають задовольняти вимогам, дотримання яких робить систему зручною у використанні:

- платформна незалежність;
- мова реалізації;
- продуктивність;
- масштабованість;
- можливість розширення й інтеграції;
- простота використання, наявність засобів розробки;
- наявність необхідних програмних бібліотек.

Розглянемо найбільш популярні нині платформи, їх особливості, а також оцінку з погляду наведених критеріїв.

2.8.2. Огляд базових технологій

Нині є безліч розроблених технологій серверної сторони, як комерційних, так і вільно поширюваних.

Платформи розглядаємо з погляду побудови на них складних гетерогенних Web-систем, тому деякі з популярних технологій не наводимо в детальному огляді через неможливість або недоцільність їх використання як базової платформи. Наприклад, технологія ISAPI та інші технології, які розширюють можливості Web-серверів, не підходять для застосування внаслідок прив’язки

до конкретного Web-сервера. Наведемо тільки основні технології, потенційно здатні створити складні гетерогенні Web-системи.

Технологія Common Gateway Interface. Технологія CGI вирізняється серед інших розглянутих технологій тим, що є найбільш низькорівневим стандартом інтерфейсу, який забезпечує зв'язок зовнішньої програми з web-сервером.

Сам протокол розроблено таким чином, щоб можна було використовувати будь-яку мову програмування та працювати зі стандартними пристроями введення/виведення. Оскільки така можливість надається на рівні операційної системи, то використовують або скрипт, написаний на відповідною мовою програмування, або простіше його подання - командний файл.

Розглянемо основні переваги та недоліки технології CGI за окремими критеріями:

– технологія CGI не передбачає особливих обмежень для використання платформи та web-сервера, тому працює на всіх популярних платформах і web-серверах, а також технологія не прив'язана до конкретної мови програмування й може бути використана будь-якою мовою, що працює зі стандартними потоками введення/виведення;

– продуктивність CGI-програм невисока, основною причиною чого є те, що у разі чергового звертання до сервера для роботи CGI-програми створюється окремий процес, який вимагає великої кількості системних ресурсів;

– технологія не передбачає вбудованих засобів масштабованості, це потребує додаткового доопрацювання програмного забезпечення розробниками;

– CGI-програма являє собою готовий до виконання файл, що перешкоджає легкому розширенню системи.

Із зазначених причин на сьогодні віддають перевагу більш розвиненим платформам, які надають більше зручності розробникам та мають підвищену продуктивність. Однак велика кількість уже розробленого програмного забезпечення змушує розвивати й використовувати технологію CGI, потрібну для розуміння принципів роботи високорівневих платформ.

Технологія Personal Home Page. Технологія PHP набула значного поширення завдяки своїй безкоштовності й підтримання найбільш популярних платформ, оскільки грунтується на принципі побудови сторінок із шаблонів, який вперше запропоновано в технології ASP. Технологія PHP розвиває і доповнює цей принцип. Сторінки PHP мають вигляд звичайних HTML-сторінок, у яких можуть використовуватися спеціальні теги *<? Php i?>*, між якими вставляються рядки програмного коду спеціальною мовою сценаріїв PHP.

Принцип шаблонів дозволив розробникам писати програми набагато швидше і без помилок, властивих традиційним CGI-програмам, які пересилають HTML-вміст у потік виведення. На сьогодні низка систем, побудованих на шаблонах, містить як прості сторінки з вибірками з бази даних, так і великі прикладні програми електронної комерції, в основі яких лежить мова XML. Системи з використанням шаблонів мають велику популярність, оскільки найбільше підходять для типових сайтів. Такі рішення розробляються за допомогою технологій ColdFusion, PHP, JSP і ASP, з яких PHP є найбільш поширеною.

Розглянемо основні переваги та недоліки платформи:

– застосовувана в PHP мова проста і зручна, однак не є, в повному розумінні, об’єктно-орієнтованою;

– для PHP наявні великі бібліотеки, а також безліч вбудованих функцій для вирішення найрізноманітніших завдань;

– у разі використання PHP з Web-сервером Apache є можливість ефективного виконання ядра як розширення сервера, інакше продуктивність платформи невисока;

– власних засобів масштабування PHP не має, всі можливості з кластеризації цілком покладено на Web-сервер;

– можливості інтеграції обмежені наявністю лише функцій включення модулів і використання зовнішніх функцій, що не відповідає сучасним вимогам;

– підхід PHP, в основу якого покладено шаблони, за умови наявності значних переваг разом з тим має і серйозні недоліки.

Із загальних недоліків цього підходу, застосовуваних як до PHP, так і ASP, JSP необхідно звернути увагу на такі:

– файл-сторінку може підтримувати тільки людина, яка добре володіє як мовами програмування, так і мовою HTML, що вимагає підвищеної кваліфікації;

– один файл у конкретний момент часу може правити лише одна людина. Це означає, що працює або програміст, або дизайнер, тобто неможливий розподіл праці між групою фахівців;

– зберігання бізнес-логіки у файлах–сторінках у розподіленому між керівними елементами вигляді ускладнює її винесення в об’єкти другого рівня.

Таким чином, завдяки простоті використання, наявності великої кількості функцій і бібліотек, поширеності й підтримки більшості наявних Web-серверів і платформ, PHP є дуже зручним засобом розробки невеликих систем. У той же час, обмеження щодо продуктивності, масштабованості, неповноти як мови програмування, а також можливостей розширення та інтеграції перешкоджають використанню платформи для розробки масштабних систем.

Технологія Java Servlets. Технологію Java Servlets (сервлети) було розроблено компанією Sun Microsystems, щоб використовувати переваги платформи Java для розв’язання проблем технології CGI та API-розширень сервера, зокрема проблеми продуктивності, виконуючи всі запити як нитки в одному процесі. Сервлети також можуть легко розділяти ресурси і не залежать від платформи, оскільки виконуються всередині Java Virtual Machine (JVM).

Технологія має широкі функціональні можливості, оскільки велика кількість бібліотек надає найрізноманітніші засоби, необхідні під час розробки. Модель безпеки Java уможливлює точне керування рівнем доступу, наприклад дозволяючи доступ тільки до певної частини файлової системи. Завдяки обробці винятків Java-сервлети стають більш надійним засобом, ніж розширення серверів на мовах програмування C/C#.

Будь-який сервлет є класом Java, тому має бути виконаний усередині JVM **сервлет-контейнером** (servlet container, servlet engine), який завантажує клас сервлета під час першого звертання до нього або відразу в момент запуску сер-

вера за спеціальною вказівкою. Далі сервлет залишається завантаженим для обробки запитів, поки він не вивантажується явно, або до зупинення контейнера.

Технологія є поширеною і може бути використана для створення програмного забезпечення, яке працює з усіма популярними Web-серверами (Enterprise Server від Netscape, Microsoft Internet Information Server (IIS), Apache, Java Web-сервер від Sun).

Програмний інтерфейс дозволяє сервлетам обробляти запити на будь-якому рівні, у разі потреби використовуючи низькорівневі дані, такі як заголовки запитів, їх тип, що надає великої гнучкості під час розробки нестандартних обробників, наприклад для роботи з двійковим або мультимедійним вмістом.

Оскільки сервлети обробляють в одному процесі за рахунок створення потоків усередині, програмний код сервлетів, який міститься у потоці, має бути безпечним. Це накладає певну відповідальність на програміста, але за допомогою стандартних прийомів, таких як відмова від використання полів у класах сервлетів і зберігання необхідних даних у контексті або зовнішньому сховищі, такі властивості коду легко досягаються. За таких умов сервлети набувають таку перевагу, як масштабованість.

Отже, сервлети забезпечують компонентний, платформо-незалежний метод для побудови web-орієнтованого прикладного програмного забезпечення, яке не має обмежень продуктивності CGI-програм. Вони мають широкий діапазон доступних прикладних API, дозволяють використовувати всі переваги Java, легко розширюються і масштабуються, підтримуються всіма популярними Web-серверами. Все це робить їх засобом розробки великих Web-систем.

Технологія Java Server Pages. Технологія JSP від компанії Sun Microsystems стала надбудовою над технологією Java Servlets та забезпечує більш швидку і просту розробку web-прикладних програм за рахунок застосування підходу, який використовує шаблони програмування.

Для розуміння архітектури і переваг JSP необхідно знати технологію Java Servlets, оскільки вони тісно пов’язані. Сторінки JSP являють собою шаблони сторінок HTML, аналогічні шаблонам PHP і ASP. Основною відмінністю від інших подібних технологій є те, що код, який міститься всередині спеціальних тегів, не інтерпретується під час звертання до сторінки,

а попередньо компілюється в Java Servlet. Статичні ділянки шаблона перетворюються на виклики до функцій для їх розміщення в потік виведення. Код компілюється так, неначе він міститься всередині сервлета. Компіляція JSP сторінок у сервлети є трудомісткою, але виконується одноразово або під час першого звертання до сторінки, або у процесі запуску сервлет-контейнера.

Технологія JSP вдало поєднує підхід з використанням шаблонів до побудови сайтів і всі переваги Java-платформи, завдяки чому набула значного поширення як для створення професійних комерційних розробок, так і для відкритих безкоштовних проектів. Важливим кроком до розширення підходу з використанням шаблонів стали бібліотеки тегів (tag libraries), які створили можливість інтегрувати стандартні, сторонні або власні програмні компоненти у сторінки. Простота створення та використання зумовила популярність бібліотек тегів.

Завдяки роботі на основі Java технологія JSP не прив'язана до конкретної апаратної або програмної платформи, тому JSP є зручним рішенням для використання в гетерогенних середовищах.

Продуктивність технології обмежена такими об'єктивними особливостями архітектури: по-перше, сторінки мають бути відкомпільованими в сервлети, що потребує багато часу; по-друге, сервлети виконуються в середовищі Java, тобто в режимі інтерпретації.

Однак ці обмеження компенсуються додатковими можливостями, оскільки сучасні контейнери підтримують кластеризацію серверів, то навантаження перекладається на апаратне забезпечення. Це є економічно виправданим і простим рішенням. Завдання компіляції в сервлети є одноразовим та виконується або під час першого звертання, або у процесі запуску сервлет-контейнера, тому не позначається на загальній продуктивності системи у разі розгляду за достатній період.

Основними перевагами JSP є простота програмування, характерна для підходу з використанням шаблонів, наявність великої кількості сторонніх бібліотек, легкість їх застосування, потужні й різноманітні середовища розробки. Завдяки цим факторам JSP є найбільш перспективною базовою технологією розробки у процесі створення Web-сайтів. Однак у разі створення

складних Web-систем обмеження, які накладаються підходом з використанням шаблонів, стають серйозною перешкодою для розвитку цієї технології.

Технологія Microsoft.NET і середовище ASP.NET. Технологія .NET є новітньою розробкою компанії Microsoft і позиціонується як новий етап у розвитку засобів взаємодії між прикладними програмами. Нині вона доступна як доповнення .NET Framework до сімейства операційних систем Microsoft Windows, у продуктах Windows Server, а також ведуться роботи зі створення .NET Framework для інших операційних систем. Платформа .NET спрощує розробку прикладних програм і підвищує надійність коду, зокрема забезпечує автоматичне керування часом життя об'єктів, незалежні від мов програмування бібліотеки класів, обробку винятків і налагодження.

Основа .NET – середовище CLR (загальне середовище виконання мов) спирається на системні служби операційної системи і керує виконанням коду, написаного будь-якою сучасною мовою програмування. Набір базових класів надає доступ до сервісів платформи, які розробники можуть використовувати будь-якою мовою програмування. Середовище CLR і базові класи разом становлять основу .NET платформи, яка пропонує такі високорівневі сервіси:

– ADO.NET – нове покоління ADO, яке використовує XML і SOAP для обміну даними;

– ASP.NET – нова версія ASP, що дозволяє використовувати будь-яку (.NET сумісну) мову для програмування Web-сторінок;

– Windows Forms і Web Forms – набір класів для побудови користувацького інтерфейсу локальних та Web-орієнтованих прикладних програм.

Розгортання систем на платформі .NET здійснюється особливим чином; вихідні коди компілюються не в команди процесора x86 або інші машинні коди, замість цього компілятор створює код мовою проміжного шару програмного забезпечення, запропонованою Microsoft (Microsoft Intermediate Language – MSIL). Файл, що містить MSIL, може виконуватися на платформі будь-якого процесора, якщо операційна система має у складі .NET CLR.

Важливою складовою частиною платформи .NET є нове середовище ASP.NET. Можливості ASP.NET значно розширено, в його основі лежить інша платформа, і базовими мовами програмування для неї обрано C# і

Visual Basic, замість колишніх скриптових мов. У той же час, нова технологія дозволяє писати ASP-сторінки будь-якою відповідною мовою.

В ASP.NET закладено все для того, щоб зробити весь цикл розробки web-орієнтованих програм швидшим, а підтримку простішою. Основні можливості й принципи роботи ASP.NET такі:

- компілювання коду під час першого звертання;
- широкий вибір бібліотек компонентів, що постачаються з .NET;
- підтримка потужного засобу розробки – Visual Studio .NET;
- мовна незалежність у межах платформ, для яких реалізовано загальне мовне середовище виконання программ – CLR;
- можливості розширення за допомогою мультипроцесорних і кластерних рішень;
- нові можливості з обробки помилок;
- об’єктно-орієнтовані мови розробки (нова мова C#);
- розширені можливості повторного використання компонент.

Очевидно, що платформи .NET і ASP.NET надали нових можливостей для розробки Web-систем, задовольняють усім сучасним вимогам й дозволяють значно прискорити і спростити розробку складних прикладних програм. Однак на сьогодні .NET у повному обсязі наявне тільки для платформи Windows, а його перенесення на інші системи перебуває на стадії розробки, але ще не завершено, і майбутні результати оцінити важко. Оскільки ASP.NET сильно прив’язана до сервера IIS, хоча архітектура .NET дозволяє перенести прикладну програму ASP.NET на іншу платформу, нині реальної можливості розробляти сайти як кросплатформні рішення немає. Таким чином, найважливіша вимога до Web-орієнтованого програмного забезпечення – багатоплатформність поки що не може бути виконана платформою .NET, тобто її використання для створення Web-системи не завжди виправдано. Однак слід відзначити, що Web-система має інтегруватися з платформою .NET (передусім з Web-сервісами), оскільки її широке використання у майбутньому не викликає сумнівів.

Порівняння технологій. Розгляд найбільш популярних базових технологій побудови прикладних програм серверної сторони дозволяє назвати такі основні особливості їх архітектури:

– окреме виконання запитів, тобто під час кожного запиту динамічного вмісту запускається певна програма для його обробки, яка генерує вміст, що передається клієнтові,- цей підхід використовується у класичних CGI-скриптах;

– накопичення виконуваних процесів – підхід, аналогічний попередньому, але, якщо запит виконується повторно, нового запуску програми не відбувається, а обробка передається існуючому процесу,- такий підхід застосовують в технологіях Java Servlets, Fast CGI;

– шаблони сторінок, тобто під час запиту шаблони заповнюються динамічним вмістом, але необов'язково, вміст інтерпретується мовою сценаріїв, цей підхід застосовують в технологіях ASP, JSP, PHP;

– розширення Web-сервера,- який звертається до особливих розширень для опрацювання динамічного вмісту, причому ці розширення специфічні для Web-сервера, цей підхід використовують в ISAPI, NSAPI, mod_perl.

Кожен із зазначених підходів має свої переваги та недоліки і, відповідно, свою сферу застосування. Модель окремого виконання запитів істотно обмежує продуктивність. Варіант моделі з накопиченням процесів є розвитком технології побудови Web-орієнтованих прикладних програм, підвищує продуктивність, зберігаючи при цьому максимальну гнучкість розробки. Підхід, який використовує шаблони, надзвичайно зручний у разі розробки невеликих систем, однак зі збільшенням складності він починає гальмувати, тому процес розробки не підходить для великих систем. Він також вирізняється невисокою продуктивністю, хоча дослідження показують, що за певних умов може демонструвати досить високі показники і конкурувати з іншими підходами. Розширення Web-сервера не є найзручнішим засобом розробки, оскільки необхідно буде жорстко прив'язувати систему до певного Web-сервера, але система при цьому демонструє максимальну продуктивність і дає найбільшу гнучкість у процесі розробки прикладних сервісів.

За схемою обробки запитів платформи поділяють таким чином (технологію CGI не розглядається через незручність у використанні, низьку ефективність і те, що розширення серверів занадто сильно прив'язані до конкретних програмних продуктів):

– PHP-шаблони, під час виконання на Web-сервері Apache використовують як розширення сервера інтерпретатор (в експериментальному режимі можна також застосовувати сервер IIS);

– Java Servlets – накопичення процесів для кожного сервлета;

– JSP-шаблони – у процесі обробки виконується їх предкомпіляція в Java Servlets, що дозволяє використовувати схему накопичення процесів;

– ASP.NET шаблони - використовується схема попередньої компіляції, а не інтерпретації коду, для цього необхідною є наявність розширення Web-сервера IIS, також можна застосовувати і низькорівневі обробники.

Порівняльну оцінку основних характеристик платформ наведено у табл. 2.2, у якій «–» – немає підтримки; «–/+» – недостатня підтримка; «+/–» – підтримка не в повному обсязі; «+» – повна підтримка. Для порівняльних характеристик, таких як мова реалізації або продуктивність, оцінки відповідають ступеню переваги технології.

Таблиця 2.2. Порівняльна оцінка основних характеристик платформ

	PHP	**Java Servlets**	**JSP**	**ASP .NET**
Багатоплатформність	+/–	+	+	–/+
Продуктивність	–/+	+/–	+/–	+
Масштабованість	–	+	+	+
Мова реалізації	+/–	+	+	+
Можливості розширення й інтеграції	–	+	+/–	+
Простота використання, наявність засобів розробки	+/–	+/–	+	+
Наявність необхідних програмних бібліотек	+	+	+	+
Поділ дизайну і логіки	+/–	–/+	+/–	+
Засоби візуальної розробки	–/+	+/–	+	+
Можливість побудови компонентної архітектури	–	+	+/–	+

2.8.3. Технологія Rich Internet Applications

Нині у сфері створення Internet-прикладних програм визначилася тенденція до переходу від стандартних технологій до платформ, які дозволяють запускати в середовищі Web-браузера програми, які за зовнішнім виглядом і поведінкою не відрізняються від настільних (desktop) програм. Такий вид програмних продуктів називають Rich Internet Applications (RIA). Незважаючи на те, що деякі з технологій RIA використовують досить давно (з кінця 90-х років), найбільшого поширення вони набули останнім часом.

Зазвичай RIA прикладна програма:

– передає Web-клієнтові необхідну частину призначеного для користувача інтерфейсу, залишаючи велику частину даних (ресурси програми, дані, тощо) на сервері;

– запускається у браузері й не вимагає установки додаткового програмного забезпечення;

– запускається локально в середовищі обчислювального процесу, в якому гарантовано безпеку, названого «пісочниця» (sandbox).

За допомогою діаграми Венна на рис. 2.52 продемонстровано можливі області застосування RIA як технології побудови програмних систем.

Рис. 2.52. Можливі області застосування RIA як технології побудови програмних систем

Традиційні Web-прикладні програми переважно працюють на сервері – у клієнтській програмі-браузері (Internet Explorer, Firefox або Opera) відображається контент, написаний мовою розмітки HTML. Згідно з цим підходом традиційні Web-прикладні програми мають такі недоліки у процесі функціонування:

мають обмежений рівень інтерактивності; постійно взаємодіють із сервером – надсилають на нього дані, отримують відповіді й завантажують нові сторінки.

У RIA значну частину функціонала винесено у клієнтську частину, тобто низка функцій виконується на стороні клієнта (у тому ж браузері, але з використанням плагинів і/або технології Ajax), що дозволяє отримати прикладну програму, яка має такі переваги: працює швидше за рахунок меншої кількості звертань до сервера і того, що немає перевантажень сторінок; забезпечує близький до традиційних настільних прикладних програм рівень інтерактивності та зручності інтерфейсу (наприклад, drag&drop).

Розглянемо архітектурні особливості побудови RIA–прикладних програм. Порівняння принципів роботи Web-прикладної програми і RIA-прикладної програми зображено на рис. 2.53.

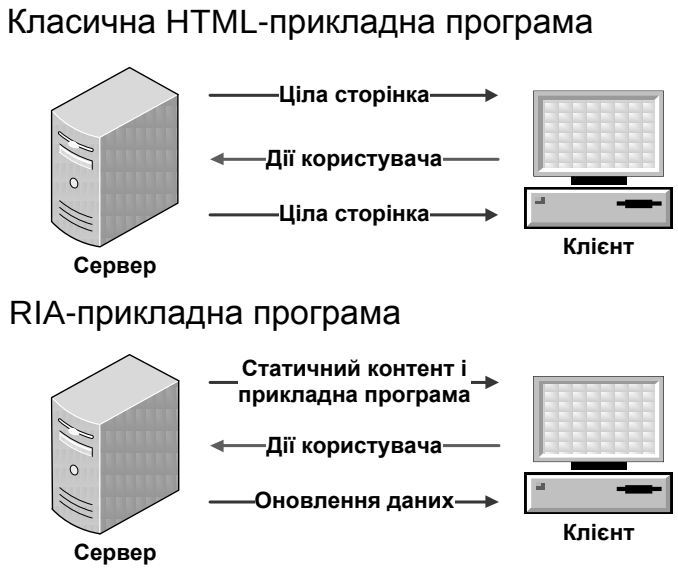


Рис. 2.53. Принципи роботи традиційної Web-прикладної програми і RIA-прикладної програми

Порівняльні дані профілізації традиційної Web-прикладної програми (на прикладі сайту mail.ru) і RIA-прикладної програми (на прикладі сайту gmail.com) подано у табл. 2.3.

Таблиця 2.3. Порівняння даних профілізації Web- і RIA-прикладних програм

Дія	**mail.ru**	**gmail.com**
Початкова сторінка	24 запити 116 КБайт	33 запити 379 КБайт
Відкрити лист	17 запитів 71 КБайт	3 запити 2 КБайт

Також можна порівняти архітектуру товстого клієнта і RIA-прикладної програми.

Товстий клієнт – це високопродуктивна інтерактивна прикладна програма з багатим, призначеним для користувача, інтерфейсом, яка працює в різних сценаріях незалежно від наявності Internet.

Зазвичай архітектуру розглядають за такими компонентами (рис. 2.54):

– рівень подання – відповідає за взаємодію прикладної програми з користувачем, формує і перевіряє введені користувачем дані;

– рівень бізнес-логіки – охоплює бізнес-потоки й логіку, які власне і керують прикладною програмою; також на цьому рівні перебувають бізнес-сутності, тобто елементи, якими обмінюються компоненти логіки;

– рівень даних – містить моделі та провайдери, які дозволяють отримувати дані.

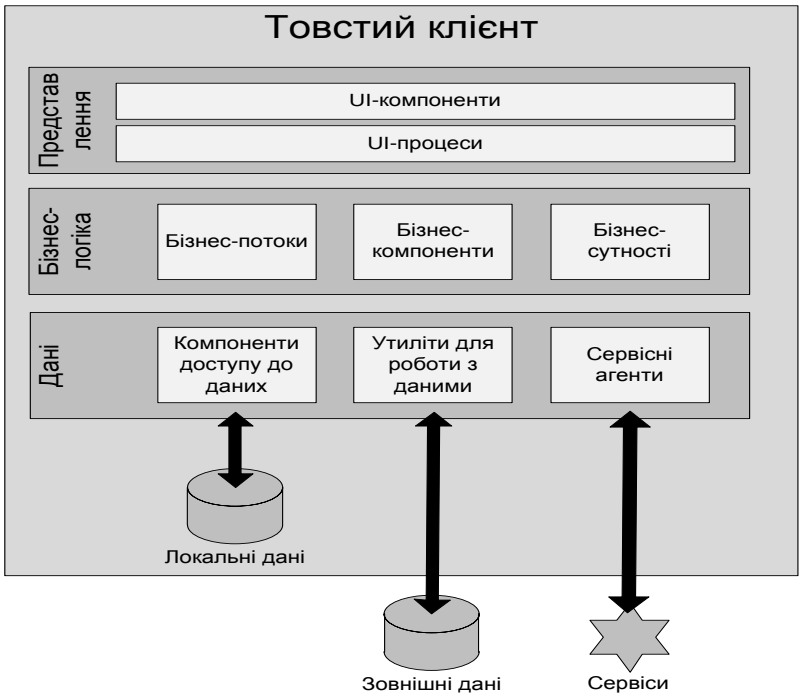


Рис. 2.54. Архітектура товстого клієнта

Internet-прикладні програми RIA виконуються в «пісочниці» браузера. Архітектура Web-сервера має такі компоненти (рис. 2.55):

– рівень сервісів надає інтерфейси для клієнтської частини для взаємодії з рівнем логіки;

– рівень бізнес-логіки ідентичний рівню логіки товстого клієнта;

– рівень даних ідентичний рівню даних товстого клієнта.

Порівняння архітектури програмного забезпечення за технологією товстого клієнта й архітектури RIA-прикладної програми подано у табл. 2.4.

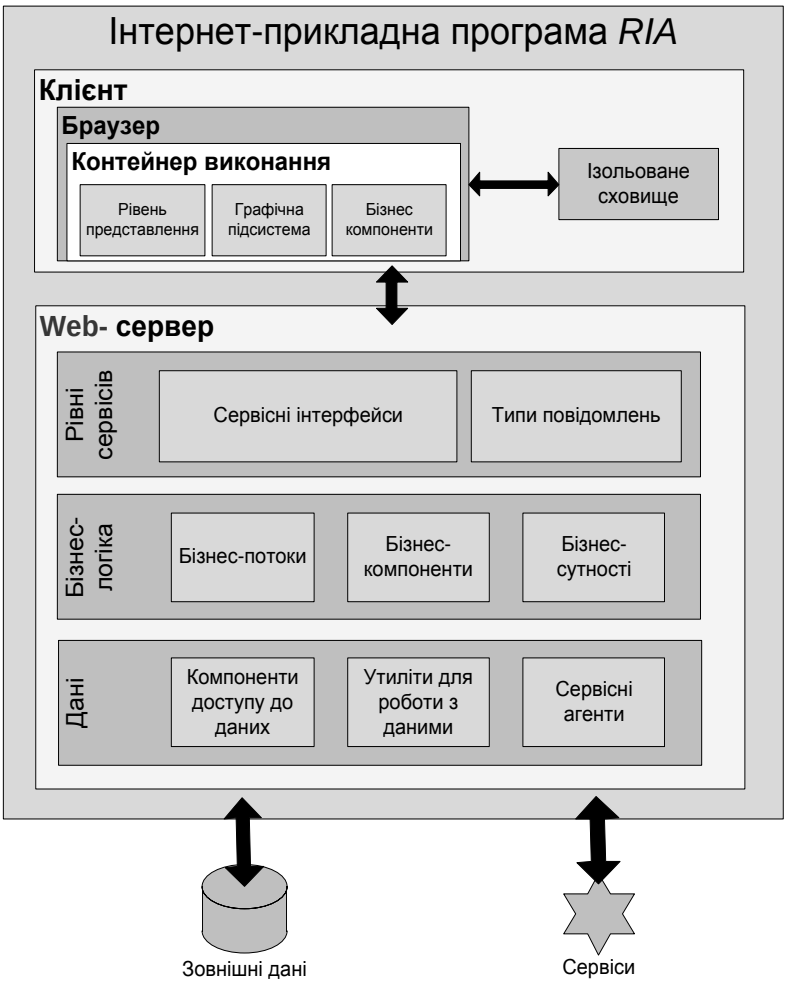


Рис. 2.55. Архітектура RIA-прикладної програми

Таблиця 2.4. Порівняння архітектури програмного забезпечення за технологією товстого клієнта й архітектури RIA-прикладної програми

Архітектура	**Переваги**	**Проблеми**
Товстий клієнт	1. Може витрачати клієнтські ресурси; 2. інтерактивність, багатий призначений для користувача функціонал; 3. висока здатність до реагування; 4. підтримка роботи без мережі.	1. Вимагає установки (Click Once, Windows Installer, Copy); 2. контроль версій неефективний або відсутній; 3. залежність від платформи
RIA	1. Такий самий багатий, призначений для користувача, інтерфейс, як і у технології товстого клієнта; 2. підтримка 3D і медіа високої якості; 3. простий спосіб поширення; 4. простий механізм контролю версій; 5. крос-платформність і крос-браузерність.	1. Невеликий розмір прикладних програм; 2. обмеження щодо використання клієнтських ресурсів; 3. вимагає встановленого плагіна для браузера.

Узагальнюючи, зауважимо, що RIA-прикладні програми залучили:

від товстого клієнта:

- надання користувачеві інтерактивного інтерфейсу;
- швидкий час відгуку інтерфейсу без оновлення сторінки;
- звичні засоби призначеного для користувача інтерфейсу, зокрема drag&drop і можливість роботи в режимі онлайн та оффлайн;

від Internet-прикладної програми:

- швидке розгортання;
- крос-платформність;
- використання прогресивного завантаження для отримання інформації і даних;
- використання більшості Internet-стандартів;

від засобів передачі даних:

- інтерактивні аудіо- і відеотехнології.

Розглянемо технології створення RIA-прикладних програм, які використовують на сьогодні.

Технології Adobe. Фірма Adobe послідовно просуває свої RIA-технології на основі Flash-платформи і надає повний стек рішень та інструментів для створення візуальної та медійної складових, а також серверних рішень. Flash-плеєр доступний на всіх основних платформах і в усіх основних браузерах. Окрім того, наявна версія для мобільних пристроїв. Стек технологій Adobe для створення RIA-прикладних програм виглядає, як показано на рис. 2.56.

Adobe Flash. Adobe Flash (раніше відома як Macromedia Flash) – мультимедійна платформа, використовувана для створення векторної анімації та інтерактивних прикладних програм (у тому числі, ігор), а також для інтеграції відеороликів у Web-сторінки.

Adobe Flash дозволяє працювати з векторною, растровою та обмежено з тривимірною графікою, а також підтримує двонаправлену потокову трансляцію аудіо і відео. Для кишенькових персональних комп'ютерів (КПК) й інших мобільних пристроїв випущена спеціальна полегшена версія платформи Flash Lite, функціональність якої обмежена з погляду можливостей мобільних операційних систем і їх апаратних характеристик.

Adobe Flex. Adobe Flex також дозволяє розробляти прикладні програми на базі Flash-платформи, але без використання часової шкали, замість якої пропонується мова розмітки MXML, що дає можливість у декларативній формі задавати зміни станів і переходи між ними. Adobe Flex призначено виключно для розробників, оскільки з його допомогою створюють складні Internet-прикладні програми для бізнес-завдань і для щоденного застосування. Уся розробка у Flex орієнтована на використання готового набору розширюваних компонентів, що підходять майже для всіх завдань. Таблиці стилів CSS дозволяє гнучко настроювати зовнішній вигляд як окремих компонентів, так і всієї прикладної програми. Зв'язування даних допомагає відображувати змінену інформацію фактично без єдиного рядка коду.

Flex-прикладна програма може компілюватися на сервері (для цього буде потрібно *mod_flex.so* або *mod_flex.dll* залежно від ОС і Web-сервера) або з IDE, або безпосередньо з командного рядка за допомогою компілятора

mxmlc (починаючи з Flex 2). Як і у Flash, результатом є файл swf, виконуваний у Flash Player. Принцип роботи Flex показано на рис. 2.57.

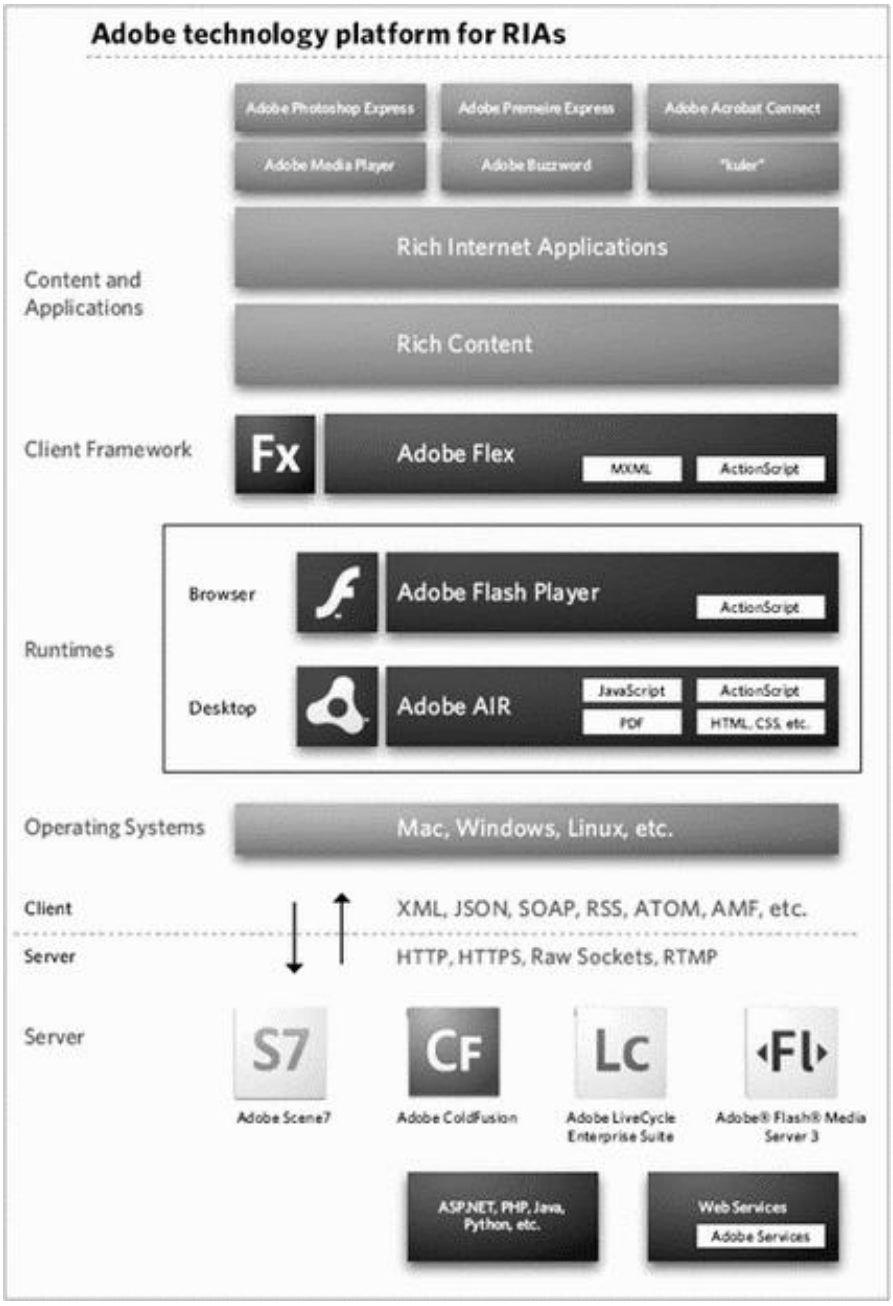


Рис. 2.56. Стек технологій Adobe для створення RIA-прикладних програм

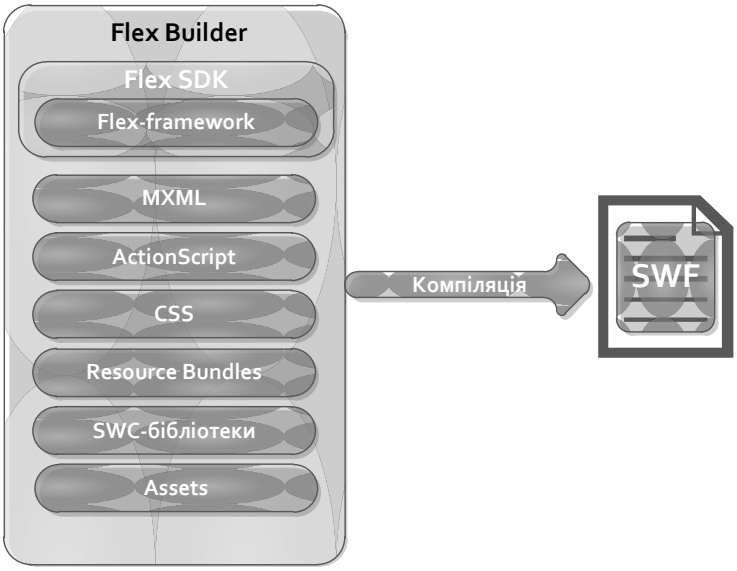


Рис. 2.57. Принцип роботи Flex

Flex – це великий набір класів (Flex SDK), що містить багато компонентів, які не увійшли у Flash. Позитивні якості Flex зумовлені його специфікою, зокрема мовою MXML, наприклад, Flex-компоненти для побудови графіків є зручними і виглядають дуже прийнятно для використання у стилі «як є».

Adobe Air. Adobe Air (Adobe Integrated Runtime) дозволяє запускати Web-прикладні програми на основі як HTML, так і Flash-платформи з можливістю перегляду PDF-документів на «робочому столі» персонального комп'ютера. Таким чином, розробники Web-орієнтованого програмного забезпечення можуть створювати настільні прикладні програми, використовуючи вже наявні типові шаблони прикладних компонентів. Adobe AIR надає низку можливостей щодо інтеграції з операційною системою: вікна, доступ до файлової системи, захищене локальне сховище, локальну базу даних і технологію перетягання об'єктів (drag&drop). AIR-прикладні програми можуть встановлюватися прямо з Internet. Архітектуру платформи AIR подано на рис. 2.58.

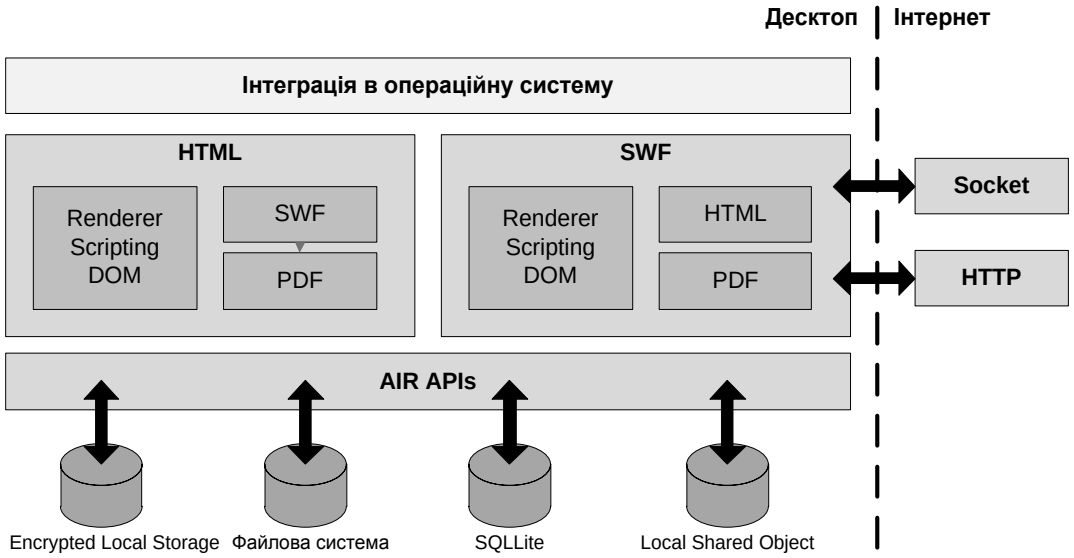


Рис. 2.58. Архітектура платформи AIR

Технологія Ajax. Ajax є поєднанням технологій, в основі яких лежать мова JavaScript і об'єкт XMLHttpRequest, завдяки чому прикладні програми можуть отримувати дані з сервера і відображати їх без перезавантаження сторінки. Формально Ajax не є RIA-технологією, оскільки не інтегрує в собі медіа можливостей, не має стандартних засобів промальовування векторної графіки, а JavaScript по-різному виконується в різних браузерах.

Ці проблеми намагаються вирішити за допомогою Ajax-бібліотек, але вони ґрунтуються на гібридних технологіях (наприклад, використання Flash для відео). Для подолання проблеми крос-браузерності також знайдено рішення, але вона знову може виникнути у разі виходу нових версій браузерів.

Нині є безліч Ajax-бібліотек, що мають свої особливості й сфери застосування, проте немає єдиного стандарту, тому, переходячи з однієї бібліотеки на другу розробник вимушений повторно вивчати документацію.

Використання Ajax дозволяє суттєво збагатити традиційні Web-сторінки і зробити користування ними дуже комфортним. Застосовувати Ajax, створюючи сайти-презентації та Internet-прикладні програми, які максимально наближені до рівня настільних прикладних програм, є недоцільним.

Основні переваги технології Ajax такі:

- наявна в будь-якому браузері;
- проста в освоєнні;

- швидкий цикл розробки;
- розвинений інструментарій;
- розвинене співтовариство.

Основні недоліки технології Ajax такі:

- іноді наявна крос-браузерна несумісність;
- складно/дорого розробляти великі прикладні програми;
- складно налагоджувати, дорого підтримувати;
- низька продуктивність.

Технології Microsoft. Корпорація Microsoft враховуючи актуальність RIA-технологій пропонує розробникам низку засобів створення насичених Internet-прикладних програм.

ActiveX. Технологія ActiveX – засіб, за допомогою якого Internet Explorer (IE) використовує інші прикладні програми всередині себе, завантажує Windows Media Player, Flash, Quicktime та інше прикладне програмне забезпечення, яке може відтворювати файли, вбудовані у web-сторінки. Елементи керування ActiveX активізуються, якщо клацнути по такому об'єкту на web-сторінці, наприклад, WMV-файлу, щоб завантажити його для відображення у вікні браузера IE.

Як продукт Microsoft ActiveX була спеціально спроектована для роботи із системами Windows, тому не підтримується іншими операційними системами, такими як Mac або Linux.

Firefox та інші крос-платформні браузери використовують систему «Програмний інтерфейс модулів Netscape» (Netscape Plugin Application Programming Interface, NPAPI), яка під'єднується та виконує функції, подібні до ActiveX.

Керівні елементи ActiveX – еквівалент модулів NPAPI, проте ActiveX і технологія модулів типу NPAPI відмінні. Наприклад, за допомогою надбудови ActiveX можна завдати шкоди комп'ютеру, модулі NPAPI мають кращі функції безпеки, менше контролюють web-сторінку, в яку вони вбудовані, та не можуть використовуватися у прикладних програмах, відмінних від web-браузеру. ActiveX має ширше застосування для великої кількості прикладних програм.

Як технологія для RIA технологія ActiveX має низку суттєвих недоліків:

- платформа тільки IE і Windows;
- нетривіальна у розробці та відладці;
- великий період освоєння технології;
- є деякі проблеми з безпекою.

Основною перевагою технології ActiveX для RIA є те, що вона дозволяє вбудовувати компоненти в різні системи без додаткового програмування.

Технологія WPF. Windows Presentation Foundation (WPF, кодова назва Avalon) – графічна (презентаційна) підсистема у складі .NET Framework 3.0, що використовує мову програмування XAML (Extensible Application Markup Language), яка дозволяє створювати широкий спектр програмних мультимедіа інтерфейсів. Технологія WPF разом з .NET Framework 3.0 працює у Windows Vista і Windows 7, а також їх можна установити у Windows XP SP2 і Windows Server 2003 і 2008.

Технологічне середовище у складі WPF і .NET Framework 3.0 призначено для користувача інтерфейсу, містить нове ядро, яке має замінити GDI і GDI+, використовувані в нинішній Windows-платформі. Технологія WPF є високорівневим об'єктно-орієнтованим функціональним шаром (framework), що дозволяє створювати 2D- і 3D-інтерфейси, у майбутньому має об'єднати Windows і Web-розробку (в тому числі, AJAX), для подання на інтерфейсі користувача використовує не GDI+, а DirectX, підтримує теми і нестандартні для WinForms контроли, її продуктивність вища, ніж у GDI+ за рахунок використання відеокарти.

Технологія Silverlight. Microsoft Silverlight надає графічну систему, схожу на Windows Presentation Foundation, і об'єднує мультимедіа, графіку, анімацію та інтерактивність в одній програмній платформі. Технологію було розроблено, щоб працювати з мовою розмітки сторінок XAML, що використовує векторну графіку й анімацію, а також із мовами Microsoft .NET. Текст, що міститься в Silverlight-прикладних програмах, доступний для пошукових систем у вигляді XAML, оскільки він не компілюється. Технологію Silverlight також можна використовувати для того, щоб створювати widget для Windows Sidebar у операційній системі Windows.

Технологія Silverlight дозволяє створювати плагіни для браузера, що запускають прикладні програми, які містять анімацію, векторну графіку та аудіо-, відеоролики, що характерно для RIA. Версія Silverlight 2.0 додала підтримку для мов .NET й інтеграцію з IDE, наступні версії (Silverlight 3, майбутня Silverlight 4) матимуть більш потужні функціональні можливості.

Silverlight підтримує відтворення WMV, WMA і MP3 для всіх підтримуваних браузерів, не вимагаючи додаткових компонентів, зокрема Windows Media Player. Silverlight дозволяє динамічно завантажувати XML-файли і використовувати програмний інтерфейс для доступу до документів DOM для взаємодії з ними аналогічно до Ajax; містить об'єкт Downloader, завдяки якому можна завантажувати скрипти, медіафайли тощо, якщо це необхідно прикладній програмі. Починаючи з версії 2.0, логіка програми може бути описана будь-якою з мов .NET, включаючи динамічні мови програмування Iron Ruby та Iron Python, які, у свою чергу, виконуються в *DLR* (*Dynamic Library Runtime*), а не *CLR*.

Технології Oracle (Sun). Розглянемо також *RIA*-технології компанії Sun Microsystems, яка входить до складу Oracle Corporation.

Технологія *JavaFX*. *JavaFX* – це платформа для створення *RIA*, які можуть запускатися на персональних комп'ютерах і мобільних пристроях. Така технологія дозволяє створювати прикладні програми для роботи з мультимедійним контентом, графічні інтерфейси користувача для бізнес-прикладних програм, ігри для персональних комп'ютерів і мобільних пристроїв, насичені графікою, мультимедіа web-сайти тощо.

Прикладна програма *JavaFX* створюється за допомогою декларативної мови програмування *JavaFX Script*, для розробки якої необхідно використовувати *JavaFX 1.0 SDK* (*Software Development Kit*).

З коду, написаного мовою *JavaFX Script*, можна звертатися до будь-яких бібліотек *Java*, тому спільне використання мов *Java* і *JavaFX Script* дозволяє вирішувати різноманітні завдання, наприклад, бізнес-логіка прикладних програм може бути написана на *Java*, а графічний інтерфейс користувача на *JavaFX Script*.

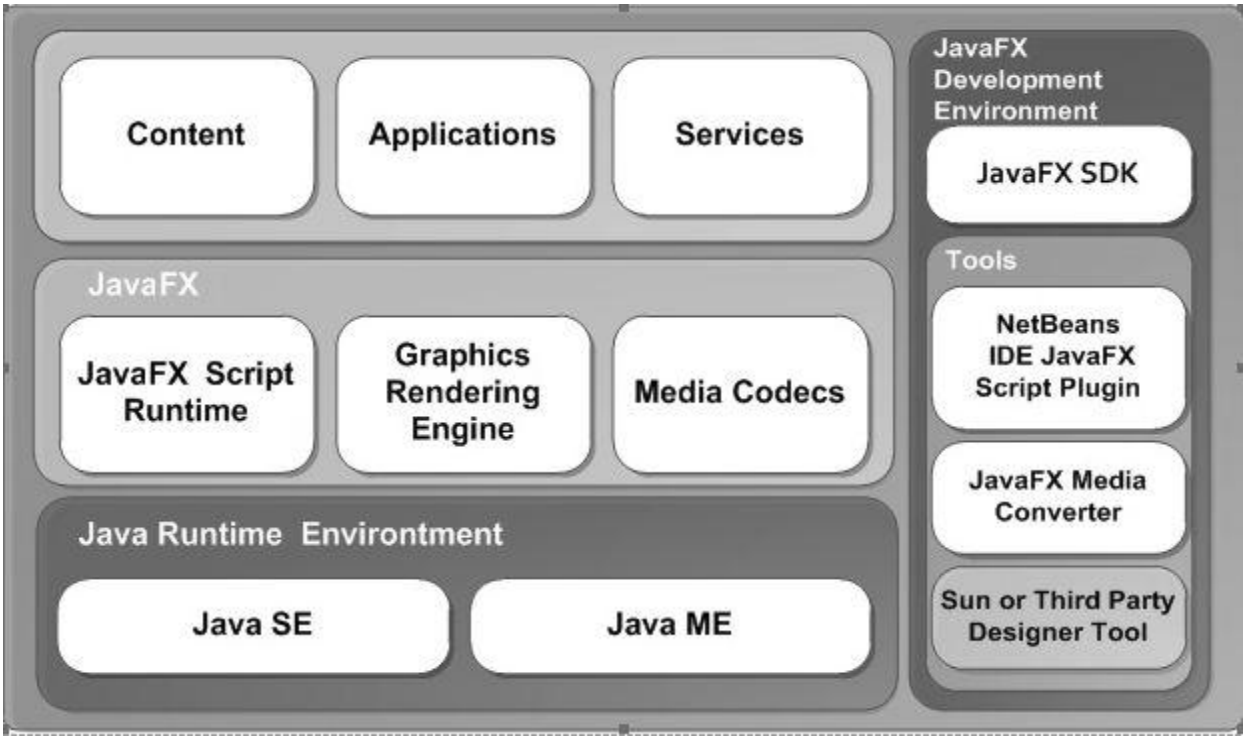


Рис. 2.59. Компоненти для створення RIA-прикладних програм

Прикладні програми, написані мовою Java FX Script, можуть запускатися на комп'ютерах зі встановленим середовищем виконання Java 1.5 і вище, нині їх виконання підтримують наступні операційні системи: Windows, Mac OS x, GNU/Linux і Solaris.

Інші технології. Наявні інші технології створення RIA-прикладного програмного забезпечення, однією з яких є Google Gears (модуль розширення для браузеру), який дозволяє AJAX-прикладним програмам частково усунути «пісочниці». Основні можливості Google Gears такі:

- доступність ресурсів offline;
- локальна SQLite база даних, синхронізація даних із сервером;
- асинхронні завдання;
- вибір декількох файлів для завантаження;
- попередній перегляд у реальному часі картинок;
- наявність технології drag&drop.

Проте восени 2009 року компанія Google зупинила розвиток технології Google Gears на користь HTML 5.0, продовжуючи тільки її підтримку.

Специфікація HTML 5.0, що перебуває на стадії розробки, підтримує такі елементи RIA:

- малювання 2D-картинок у реальному часі;
- контроль над програванням медіафайлів;
- зберігання даних у браузері;
- використання технології drag&drop;
- роботу з мережею, push-технологію (технологію просування контенту).

Порівняння вищеописаних технологій. Порівняння технологій створення RIA-прикладного програмного забезпечення наведено у табл. 2.5.

Таблиця 2.5. Порівняння технологій створення RIA-прикладних програм

Технології	**WPF**	**Silverlight**	**Flex**	**Flash**	**Adobe Air**	**JavaFX**
Основні характеристики						
ОС	Windows	Windows MacOS Linux	Windows MacOS Linux	Windows MacOS Linux	Windows MacOS Linux	Windows MacOS Linux
Браузери	IE, Firefox	IE, Firefox, Safari, Chrome	IE, Firefox, Safari, Opera, Netscape, Chrome	IE, Firefox, Safari, Opera, Netscape, Chrome	–	IE, Firefox, Safari, Opera, Netscape, Chrome
Відсутність потреби в ліцензуванні	+/–	–	+/–	–	+/–	+
Розробка						
Мови	XAML+ CLS (C#, VB)	XAML+ JavaScript/ CLS(C#, VB)/DLR (Python)	MXML+ ActionScript	ActionScript	MXML/ ActionScript /JavaScript/ HTML	JavaFX Script, Java platform (Java, Groovy)

Продовження табл. 2.5

Технології	**WPF**	**Silverlight**	**Flex**	**Flash**	**Adobe Air**	**JavaFX**
Data Binding	+	+	+	–	+	+
Декларативна мова розмітки	+	+	+	–	+	+
Змішування розмітки і коду	+	+	+	–	+	+
Code Behind	+	+	+	–	+	+
Інструментарій						
Інструментарій	Visual Studio, Microsoft Expression (Blend), XamlPad, Notepad.	Visual Studio, Microsoft Expression (Blend), Notepad	Flex Builder, Thermo, Notepad.	Flash, Flash Develop, FDT	Flex Builder, Aptana Studio, Notepad	NetBeans, Eclipse, JavaFX Pad, JFXBuilder, Notepad
WYSIWYG - редактори	Expression Blend, Visual Designer for WPF	Expression Blend	Flex Builder, Thermo	Flash	Flex Builder, Thermo, Flash	JFXBuilder
Платформи для розробки	Windows	Windows	Windows MacOS Linux	Windows MacOS	Windows MacOS Linux	Windows MacOS Linux
Вартість SDK	Безкоштовно	Безкоштовно	Безкоштовно	платна	Безкоштовно	Безкоштовно

Продовження табл. 2.5

Технології	**WPF**	**Silverlight**	**Flex**	**Flash**	**Adobe Air**	**JavaFX**
Фреймворки (каркаси)						
GUI - фреймворк	Компоненти і layout	Компоненти layout	Компоненти і layout	Компоненти	Компоненти і layout	Компоненти і layout
Розширюваність	+	+	+	+	+	+
Технологія drag&drop	+	+	+	–	+	+
Компоненти сторонніх розробників	+	+	+	+	+	–
Підтримка accessibility	+	+	+	+	+	+
GUI Automation	+	+	+	+	–	+
Кастомізація/скінизация (використання алгоритмів кешування)						
Кастомізація зовнішнього вигляду компонент	+	+	+	+	+	+
Designtime skinning (styling)	+	+	+	+	+	+
Runtime skinning (styling)	+	+	+	–	+	+
Відео						
Кодеки	(Windows Media Player)	WMV, VC - 1, H.264, ACC Video, DRM	Sorenson Spark, On2 VP6 - E, On2 VP6 - S, H.264, flv	Sorenson Spark, On2 VP6 - E, On2 VP6 - S, H.264, flv	Sorenson Spark, On2 VP6 - E, On2 VP6 - S, H.264, flv	–

Продовження табл. 2.5

Технології	**WPF**	**Silverlight**	**Flex**	**Flash**	**Adobe Air**	**JavaFX**
Hardware Acceleration	+	+	+	+	+	-
HD-відео	+	+	+	+	+	-
DRM	+	+	–	–	–	-
Streaming	+	+	+	+	+	-
Звук						
Формати	Windows Media Player	MP3, WMA, DRM	MP3, AAC	MP3, AAC	MP3, AAC	AU, AIFF, WAV, MIDI
VoIP	–	–	–	–	–	–
Наявність VoIP-рішень	+	–	+	+	+	–
Графіка						
Вбудована підтримка 3D	+	+	–	–	–	–
Векторна графіка	+	+	+	+	+	+
Завантаження JPG	+	+	+	+	+	+
Завантаження PNG	+	+	+	+	+	–
Завантаження GIF	+	–	+	+	–	+
Hardware acceleration	+	+	–	–	–	+
Бітмап-ефекти	+	+	+	+	+	+

Закінчення табл. 2.5

Технології	WPF	Silverlight	Flex	Flash	Adobe Air	JavaFX
Додатково						
Підтримка мікрофона	+	–*	+	+	+	–
Підтримка вебкамери	+	–*	+	+	+	–
Повноекранний режим	+	+	+	+	+	–
Клієнт/ сервер протоколи	.NET-сумісне програмне забезпечення	XML (JSON, Web services, RSS, POX, REST), WCF Services	HTTP, Socket, XML, AMF, RTMP	HTTP, Socket, XML, AMF, RTMP	HTTP, Socket, XML, AMF, RTMP	Програмне забезпечення, яке виконується на Java-платформі

* Очікується у версії Silverlight 4.0.

Колонка для WPF стосується XBAP, а колонка JavaFX-аплетів на основі JavaFX, оскільки розглянуті види прикладних програм орієнтовані на Web, а JavaFX- і WPF-прикладні програми можуть бути, а можуть і не бути пов'язаними з Web.

2.8.4. Дескриптор розгортання web-прикладних програм та компонент

Дескриптор (англ. Descriptor) – дослівно описувач, описовий елемент. **Дескриптор** – невід'ємне ціле число, яке задає номер будь-якого ресурсу в процесі роботи з ним і використовується зазвичай через деякий інтерфейс, причому смисл значення дескриптора схований за цим інтерфейсом.

Платформа Java 2 Enterprise Edition. J2EE-платформа – це розподілена комп'ютерна платформа, що полегшує дизайн, розробку, компонування і розгортання компонентно-орієнтованих корпоративних прикладних програм.

Розрізняють безліч способів, за якими можна подавати специфічний набір бізнес-вимог, логічну модель і опис рішення, які задовольняють таким вимогам. J2EE-платформа найбільш придатна і найкраще підходить для підтримування крос-платформності, безпеки, транзакційних прикладних програм, які надають корпоративну інформацію для Internet- та Intranet-клієнтів.

J2EE-прикладна програма являє собою набір програмних компонентів, створених для поширення на кілька рівнів. J2EE-платформа сервер-орієнтована, тому для J2EE-прикладних програм типово пропонувати засоби різноманітним клієнтам. Терміни «клієнт» і «сервер» використовують, щоб відобразити логічну, а не апаратну структуру.

Не потрібно, щоб J2EE-прикладна програма була розділеною, але вона має бути простою у використанні різними типами комп'ютерних систем, якщо це потребують ділові або технічні вимоги. Процеси розробки, компонування та поширення логічно розділені всередині середовища J2EE. Процес компонування програми керується і проводиться за рахунок дескрипторів, що містять декларативні визначення, які в підсумку формують поведінку прикладних програм у специфічному операційному середовищі.

Платформа J2EE призначена передусім для розробки розподіленого Web-орієнтованого прикладного програмного забезпечення і підтримує такі види компонентів:

Enterprise JavaBeans (EJB). Компоненти EJB призначені для реалізації на їх основі бізнес-логіки програми та операцій над даними. Будь-які компоненти, розроблені на Java, називають бінами.

Web-компоненти (Web components). Ці компоненти надають інтерфейс до корпоративних програмних систем зверху широко використовуваних протоколів Internet, зокрема HTTP. Надані інтерфейси можуть бути як інтерфейсами для людей (WebUI), так і спеціалізованими програмними інтерфейсами, що працюють подібно віддаленому виклику методів, але зверху HTTP. До групи Web-компонентів входять фільтри (filters), обробники Web-подій (web eventlisteners), сервлети (servlets) і серверні сторінки Java (JSP).

Звичайні програми на Java. J2EE (Java Platform Enterprise Edition) є розширенням J2SE (Java Platform Standart Edition), тому всі прикладні програ-

ми, написані мовою Java, можуть працювати і в цьому середовищі. Однак, окрім звичайних можливостей J2SE, ці прикладні програми можуть використовувати у своїй роботі Web-компоненти і EJB як безпосередньо, так і віддалено, зв'язуючись з ними по HTTP.

Аплети (applets). Це невеликі компоненти, що мають графічний інтерфейс користувача і призначені для роботи всередині стандартного Web-браузера. Вони використовуються тоді, коли не вистачає можливостей подання інформації для користувача інтерфейсу на основі HTML, і можуть зв'язуватися з віддаленими Web-компонентами, які працюють на сервері, по HTTP.

Компоненти, які підтримує ця платформа, мають дескриптор розгортання (deployment descriptor) – опис у встановленому форматі на основі XML- конфігурації компонента у межах контейнера, в якому він міститься. Прикладна програма в цілому також має дескриптор розгортання. Дескриптори розгортання відіграють важливу роль, дозволяючи змінювати деякі параметри функціонування компонента і прив'язувати їх до параметрів середовища, у межах якого компонент працює, не змінюючи його коду.

Розглянемо способи вирішення спільних завдань побудови розподілених систем на основі платформи J2EE.

Цілісність і несуперечність даних під час роботи J2EE-прикладних програм підтримується за допомогою механізму розподілених транзакцій, керувати якими може EJB-контейнер, який створюється визначенням політики участі методів EJB-компонентів у транзакції в їх дескрипторах розгортання або може здійснюватися вручну. В обох випадках використовуються механізми, що реалізують інтерфейси керування транзакціями Java (Java Transaction API, JTA).

Базові інтерфейси JTA містяться у пакетах *javax.transaction* і *javax.transaction.xa*. Це, передусім, інтерфейси менеджера транзакцій *TransactionManager*, самих транзакцій *Transaction* і *UserTransaction* та інтерфейс синхронізації *Synchronization*, що дозволяє отримувати повідомлення про початок завершення і кінець завершення транзакцій.

Методи інтерфейсу *TransactionManager* дозволяють запустити транзакцію, завершити її успішно або відкотити, а також отримати об'єкт, що подає поточну транзакцію і має тип *Transaction*. Методи інтерфейсу *Transaction* дають

змогу завершити або відкотити транзакцію, яку подає об'єкт такого інтерфейсу, зареєструвати об'єкти для синхронізації у процесі завершення транзакції, а також додати деякі ресурси до числа учасників даної транзакції або видалити їх із цього списку. Такі ресурси подаються у вигляді об'єктів інтерфейсу *javax.transaction.xa.XAResource*.

Інтерфейс *UserTransaction* використовують, щоб керувати призначеними для користувача транзакціями – він надає дещо менше можливостей, ніж *TransactionManager*.

У тому разі, якщо керування транзакціями цілком доручається EJB-контейнеру (це транзакції, керовані контейнером, *container managed transactions*), впливати на їх перебіг можна, зазначаючи в дескрипторах розгортання EJB-компонентів різні транзакційні атрибути (*transaction attributes*) для їх методів. Транзакційний атрибут може набувати одною з таких значень:

Required. Метод, у якого наявний такий атрибут, завжди має виконуватися в контексті транзакції, тобто він працюватиме в контексті тієї самої транзакції, в якій працював метод, що викликав його, а якщо він був викликаний поза контекстом транзакції, з початком його роботи буде запущена нова транзакція. Цей атрибут використовується найбільш часто;

RequiresNew. Метод, який має такий атрибут, завжди буде запускати нову транзакцію на самому початку роботи, при цьому зовнішня транзакція, якщо вона виконувалась, буде тимчасово припинена;

Mandatory. Метод, у якого наявний такий атрибут, має викликатися тільки з транзакції, у контексті якої він і продовжить працювати. У разі виклику такого методу ззовні транзакції буде створена виняткова ситуація, зокрема *TransactionRequiredException*;

NotSupported. У разі виклику такого методу зовнішня транзакція, якщо вона є, буде тимчасово припинена, а якщо її немає, то нова транзакція не буде запущена;

Supports. Такий метод працює у транзакції, якщо його викликали з її контексту, якщо ж він був викликаний поза транзакцією, то нова транзакція не запускається;

Never. У разі виклику такого методу з транзакції створюється виняткова ситуація *RemoteException*. Він може працювати, тільки якщо його викликано ззовні транзакції.

Відкотити автоматично керовану транзакцію можна, створивши виняткову ситуацію *javax.ejb.EJBException* або викликавши метод *setRollbackOnly()* інтерфейсу *javax.ejb.EJBContext*.

Захищеність J2EE-програми підтримується декількома способами:

1. За допомогою методів аутентифікації, тобто визначення ідентичності користувачів, які позначають у дескрипторі розгортання програми. Можна використовувати такі способи аутентифікації:

– аутентифікація не виконується;

– за допомогою основного механізму протоколу HTTP. У разі спроби звертання до ресурсу за протоколом HTTP буде запитане ім'я користувача та пароль, які перевірить Web-сервер. Цей спосіб не дуже добре захищений, оскільки реквізити користувача пересилаються мережею в незашифрованому вигляді;

– за допомогою дайджесту (digest). Цей метод працює так само, як основний механізм аутентифікації по HTTP, але ім'я і пароль користувача пересилаються в зашифрованому вигляді. Такий спосіб використовується досить рідко;

– за допомогою спеціальної форми. У такому разі використовують сторінку, на якій розміщена форма аутентифікації (зазвичай це ті самі поля для введення імені користувача та пароля, але, можливо, і якихось інших його атрибутів), і сторінка, на якій міститься повідомлення, що видається у разі невдалої аутентифікації;

– з використанням сертифіката клієнта. Цей метод використовує протокол HTTPS, клієнт повинен надати свій сертифікат або відкритий ключ, який відповідає стандарту X.509 на інфраструктуру відкритих ключів. Можна використовувати і взаємну аутентифікацію – у цьому разі й клієнт, і сервер надають свої сертифікати.

2. За допомогою з'єднань за протоколом HTTP поверх рівня захищених сокетів (Secure Socket Layer, SSL, на HTTP поверх SSL часто посилаються за допомогою окремої абревіатури HTTPS). Можна використовувати лише такі комбінації, вказавши атрибути *CONFIDENTIAL* та/або *INTEGRAL* у де-

скрипторі розгортання програми. Перший атрибут означає, що передачу даних між клієнтом та програмою буде зашифровано так, що їх важко буде прочитати третій стороні. Другий атрибут означає, що ці дані супроводжуватимуть додатковою інформацією, яка гарантує їх цілісність, тобто те, що їх не було підмінено десь між сторонами, які беруть участь у зв'язку.

3. За допомогою механізму опису ролей визначення доступності різних методів Web-компонентів і EJB-компонентів для різних ролей, а також завдання політики перенесення або створення ролей під час спільної роботи кількох методів. Ролі, політики їх перенесення і правила доступу різних ролей до методів описуються в дескрипторах розгортання компонентів. У процесі розгортання програми зареєстровані на J2EE-сервері користувачі та групи користувачів можуть бути відображені на різні ролі, зазначені в прикладній програмі.

4. За допомогою встановлення обмежень доступу до наборів ресурсів, що задаються у вигляді списків уніфікованих ідентифікаторів ресурсів (URI) або шаблонів URI. Ці обмеження описуються в дескрипторі розгортання програми та визначають ролі й дозволені їм види прямого доступу (не через звертання до інших компонентів) до певного набору URI.

5. За допомогою програмного визначення ролей і користувачів, від імені яких працює поточний потік, з коду самих компонентів. Це можна робити за допомогою методів *isUserInRole()* та *getUserPrincipal()* інтерфейсу HTTP ServletRequest, використовуваного для подання запитів до Web-компонентів, й аналогічних методів *isCallerInRole()* та *getCallerPrincipal()* інтерфейсу EJBContext, для опису контексту виконання методів EJB-компонентів.

Платформа .NET. Середовище .NET призначено для більш широкого використання, ніж платформа J2EE. Однак його функціональність у частині, призначеній для розробки розподілених Web-прикладних програм, дуже схожа на J2EE. Роль дескрипторів розгортання відіграють конфігураційні файли, подані в певному форматі на основі XML.

EJB-контейнер. У цілому EJB-контейнер являє собою приклад об'єктного монітора транзакцій – програмного забезпечення проміжного рівня, що підтримує в межах об'єктно-орієнтованої парадигми віддалені виклики методів та розподілені транзакції. Це компонентне середовище для

компонентів Enterprise JavaBeans, яке підтримує автоматичну синхронізацію Java об'єктів з базою даних.

EJB-контейнер підтримує такі базові служби при роботі з компонентами EJB:

– автоматична підтримка звертань до компонентів, розміщених на різних машинах;

– автоматична підтримка транзакцій;

– автоматична синхронізація стану баз даних та відповідних компонентів EJB у обидва боки;

– автоматична підтримка захищеності за рахунок аутентифікації користувачів, перевірки прав користувачів або компонентів на виконання викликаних ними операцій та авторизації відповідних дій;

– автоматичне керування життєвим циклом компонента (послідовністю переходів між станами «немає» – «ініціалізований» – «активний») і набором компонентів аналогічно до ресурсів, тобто видалення компонентів, що стали непотрібними; завантаження нових компонентів; балансування навантаження між наявними компонентами.

WEB-контейнер. Компонентним середовищем для роботи Web-компонентів є Web-контейнер, що постачається в межах будь-якої реалізації платформи J2EE. Web-контейнер реалізує такі служби, як керування життєвим циклом компонентів і набором компонентів аналогічно до ресурсу, тобто розпаралелювання незалежних робіт, виконання віддалених звертань до компонентів, підтримка захищеності за допомогою перевірки прав компонентів і користувачів на виконання різних операцій.

Результати роботи компонентів EJB перетворюються Web-компонентами в динамічно генеровані HTML-сторінки, і надсилаються назад до користувача, постаючи перед ним у вікні браузера.

Роль Web- і EJB-контейнерів. Процеси та синхронізація. Розбиває прикладну програму J2EE на низку процесів із потоками керування, які взаємодіють, Web- або EJB-контейнер автоматично. На їх роботу можна впливати, конфігуруючи як J2EE-сервер у цілому, так і конкретні прикладні програми. Усі методи допоміжних класів, які використовують Web-компоненти й компоненти EJB, слід оголошувати синхронізованими. Компоненти J2EE, що

працюють у межах контейнерів, можуть створювати власні окремі потоки, але робити це потрібно з великою обережністю, оскільки цими потоками контейнер керувати не зможе, і вони можуть зашкодити роботі інших компонентів.

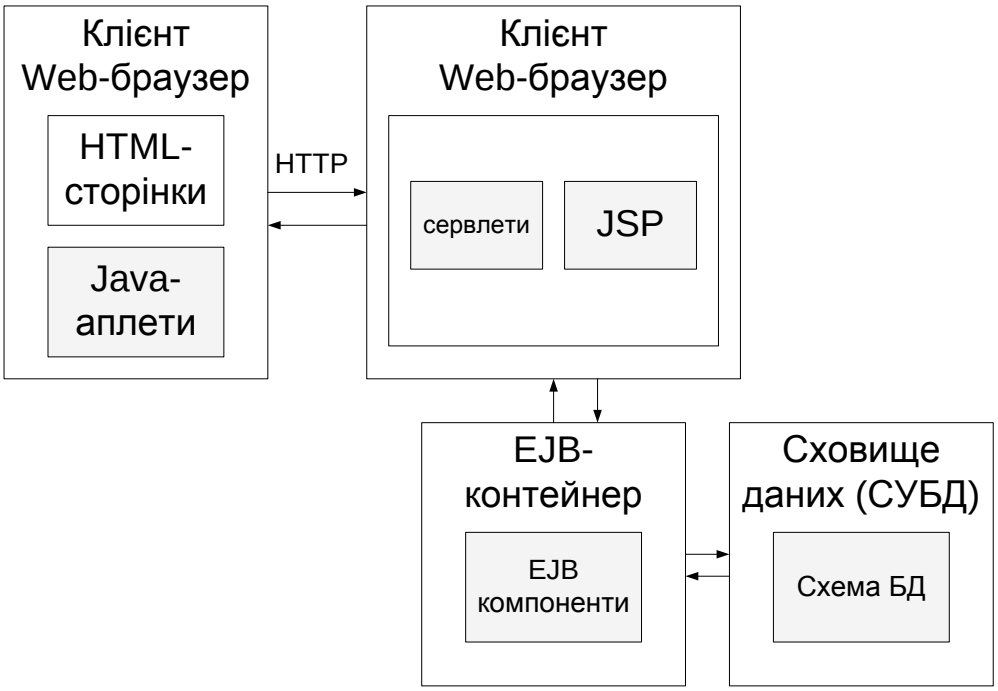


Рис. 2.60. Архітектура прикладної програми J2EE

Деякі сценарії прикладних програм та роль Web- і EJB-контейнерів. Нижче розглянуто лише кілька сценаріїв прикладних програм. Специфікація J2EE і відповідні технології мають тенденцію приймати і підтримувати різноманітність, а сценарії прикладних програм визначають, які точно APIs збираються використовуватися, щоб надати функціональність рівня прикладного програмного забезпечення. Вибір рівня реалізації прикладної програми – вибір оптимального рішення між функціональним різноманіттям і складністю. J2EE-модель програмування потрібна, щоб розглянути сценарії прикладних програм, які використовують Web-контейнер та EJB-контейнер як додаткові логічні сутності. Деякі ключові сценарії, включаючи ті, в яких Web-контейнера або EJB- контейнера, а можливо й обох, немає, відображено на рис. 2.61. Проста прикладна програма відображає свою багаторівневу модель. Це визначення передбачає наявність як Web-контейнера, так і EJB-контейнера.

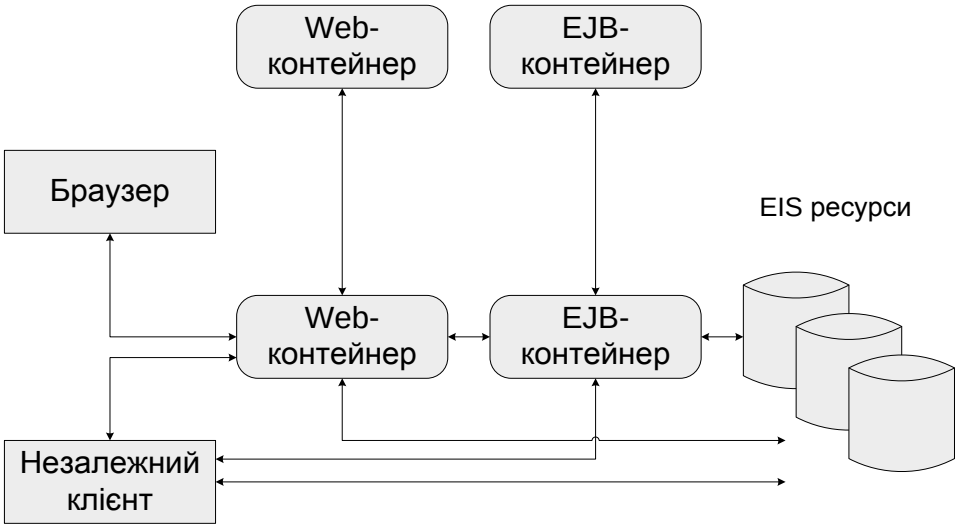


Рис.2.61. Сценарії

На вибір сценарію створення прикладного програмного забезпечення суттєво впливають такі корпоративні вимоги:

– необхідність швидко й часто змінювати вигляд прикладної програми;

– необхідність поділяти прикладні програми на представлення (інтерфейс користувача) та бізнес-логіку, щоб збільшити модульність;

– необхідність спрощувати процес подання інформації користувачам, які виконують завдання разом, причому робота кожним з них має виконуватися порівняно незалежно, але пов’язаними сценаріями;

– необхідність мати розробників, які знаються на офісних прикладних програмах, звільнених від розробки GUI і дизайну, в яких вони не можуть бути високо кваліфікованими;

– необхідність мати певний словник, щоб передати бізнес-логіку командам розробників, які розуміють вплив людського фактора й естетики прикладного програмного забезпечення;

– здатність створювати офісні прикладні програми, використовуючи компоненти з різних джерел, включаючи наявні компоненти бізнес-логіки;

– здатність розгортати виконувані (transactional) компоненти через численні апаратні та програмні платформи, незалежно від основної технології бази даних;

– здатність перетворювати внутрішні дані на зовнішні без додаткової інформації про споживача даних і виконувати це слабкозв'язаним способом.

Безсумнівно, врахування не в повному обсязі деяких або всіх названих вимог може впливати на рішення щодо проектування рівнів прикладної програми. Модель програмування J2EE вирішує проблему шляхом створення трирівневої прикладної програми так, щоб у подальшому перехід на багаторівневу архітектуру спрощувався за рахунок повторно використовуваних компонентів. Хоча розумно говорити про «недовговічність» логіки представлення (інтерфейс користувача часто змінюють), все ж таки існує порівняно мало змінюваний інтерфейс між рівнями прикладного програмного забезпечення, який зв'язує інтерфейс користувача з бізнес-логікою. Такий підхід широко використовують у разі організації взаємодії схем баз даних і даних. Нині за рахунок використання єдиного середовища доступу до ресурсів (EIS) з'являється можливість частої зміни коду прикладної програми, тобто довговічність прикладної програми значно знижується. Таким чином, модель програмування J2EE прискорює розвиток, сприяє повторному використанню компонентно-орієнтованого коду та є важелем для посилення міжрівневої взаємодії, яка використовує інтеграцію рівнів у моделі програмування J2EE.

Безліч сценаріїв функціонування прикладних програм, які продукт J2EE здатний підтримувати, ілюструє рис. 2.61, причому немає повної переваги одного сценарію над другим. У той же час продукт J2EE дозволяє підтримувати деякі або всі сценарії, що є перевагою, враховуючи те, що сценарії індивідуальні та розробляються з використанням технологій та протоколів, актуальних для розробника програми.

Багаторівневий сценарій програми. Сценарій програми, в якому Web-контейнер містить Web-компоненти, які майже повністю обробляють логіку представлення цієї прикладної програми, ілюструє рис. 2.62. Передача динамічного Web-вмісту клієнта є завданням JSP-сторінок (підтримується сервлетами). EJB-контейнер містить компоненти, які, з одного боку, відповідають на запити з Web-рівня, а з другого, мають доступ до ресурсів EIS.

Здатність ізолювати дані, що йдуть від кінцевого користувача,– перевага такого сценарію, оскільки прикладна програма неявно масштабована, але більш важливо, що офісна функціональність програми деякою мірою ізолюється від «look and feel» кінцевого користувача. Такий підхід має сенс, враховуючи що XML включений як невід'ємна частина цього сценарію.

Рис. 2.62. Багаторівневий сценарій функціонування програмного забезпечення

Здатність одночасно генерувати й опрацьовувати дані у форматі XML у Web-контейнері можна розглядати як дуже гнучку взаємодію з різними клієнтськими платформами, складність яких може бути різною, починаючи від універсального XML-підтримувального браузера закінчуючи спеціалізованою XML-генерувальної машини, орієнтованої на вертикальні рішення.

Незалежно від сфери застосування XML-дані передають через протокол HTTP. Терміном «передача даних XML» позначають модель програмування, в якій XML використовують для обміну інформацією замість застосування для цього об’єктної моделі, протилежної об’єктній моделі Java. Отже, XML можна розглядати як доповнення до мови Java.

На Web-рівні часто виникає запитання, що використовувати: JSP-сторінки або сервлети. Модель програмування J2EE використовує JSP-технологію, як основний засіб програмування у Web-контейнері. JSP-сторінки залежать від функціональності сервлетів, але модель програмування J2EE визначає JSP-сторінки як найбільш ефективний інструмент програмування для Web-інженерів, тому в Web-контейнері, спроектованому для ство-

рення динамічного вмісту для Web-клієнтів, використання технології JSP слід розглядати як норму, а використання сервлетів як виняток.

Сценарій незалежного клієнта. Сценарій, який використовує незалежного клієнта у процесі функціонування прикладної програми, показано на рис. 2.63.

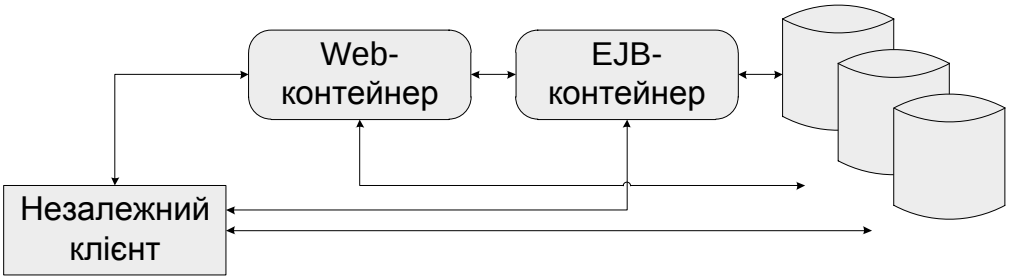


Рис. 2.63. Сценарій з використанням незалежного клієнта

У моделі програмування J2EE розглядають три типи незалежних клієнтів:

1. EJB-клієнти, які безпосередньо взаємодіють з EJB-сервером, причому корпоративні біни містяться в EJB-контейнері. Такий сценарій показано на рис. 2.64, у цьому сценарії допускається використання RMI-IIOP і EJB-сервера, який буде звертатися до ресурсів EIS за допомогою JDBC (або конекторів), показано на рис. 2.64

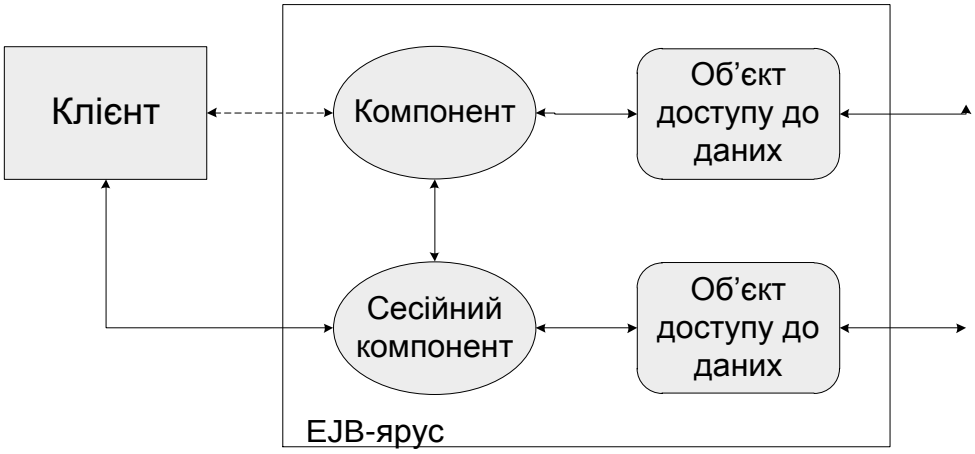


Рис. 2.64. EJB-клієнти безпосередньо взаємодіють з EJB-сервером

2. Незалежні Java-клієнти, які звертаються до ресурсів EIS безпосередньо, використовуючи JDBC або конектори. У цьому сценарії логіка представлення та бізнес-логіка за призначенням розміщені на клієнтській платформі й можуть бути інтегровані в одну прикладну програму. Цей сценарій перемі-

щує середній рівень на платформу клієнта й, по суті, є клієнт-серверним сценарієм програми, з усіма притаманними йому проблемами поширення, підтримки та оновлення.

3. VB-клієнти, які використовують динамічний Web-контент, що подається найчастіше у вигляді XML-даних. У цьому сценарії Web-контейнер керує XML-перетвореннями й забезпечує Web-зв'язок з клієнтами. Логіка представлення передається для керування на клієнтський рівень, а Web-рівень керує бізнес-логікою і прямим доступом до ресурсів EIS. В ідеалі, бізнес-логіка виноситься на EJB-сервер, де розвинена компонентна модель може її значно підсилити.

Сценарій трирівневої Web-прикладної програми. EJB-сервер є достатньо потужним інструментом для створення розподіленого програмного забезпечення. Специфікація J2EE не зобов'язує використовувати дво-, три- або багаторівневі моделі прикладних програм, тобто потрібно використовувати інструменти, які відповідають складності задачі. Сценарій трирівневої Web-прикладної програми нині дуже поширений; Web-контейнер фактично містить логіку представлення і бізнес-логіку, а за допомогою JDBC можна отримати доступ до ресурсів EIS. Сценарій функціонування трирівневої Web-прикладної програми подано на рис. 2.65.

Рис. 2.65. Сценарій трирівневої Web-прикладної програми

Використання Web-контейнера у сценарії функціонування Web-орієнтованих прикладних програм подано на рис. 2.66, у якому, варто враховувати, що термін «Web-контейнер» використано в дуже вузькому значенні.

Наприклад, якщо такий продукт J2EE вибрано для реалізації, то у Web-прикладній програмі Web-контейнер і EJB-контейнер розміщені разом (тобто міжконтейнерна взаємодія оптимізована та деталі реалізації закриті), модель програмування J2EE використовує компоненти, встановлені на такій самій платформі, як за багаторівневим сценарієм.

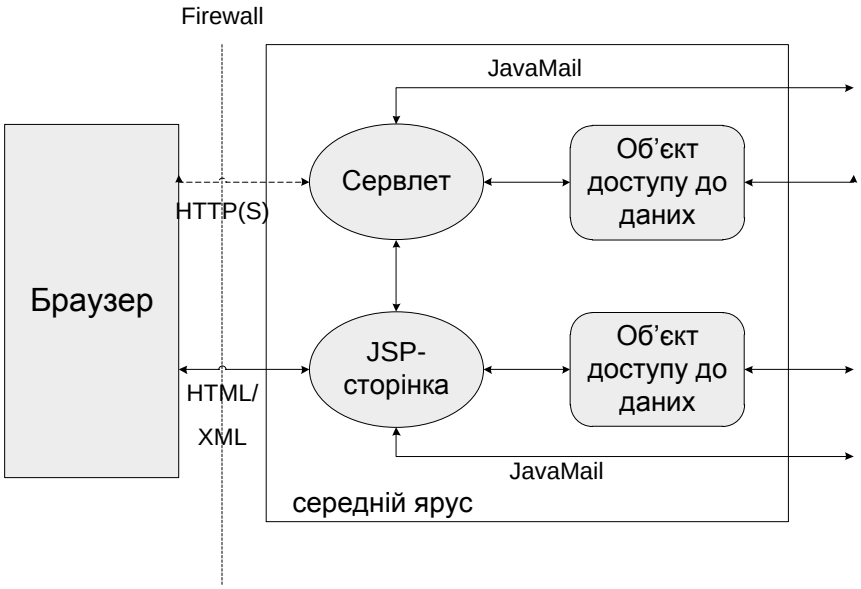


Рис. 2.66. Web-контейнер у сценарії функціонування Web-прикладних програм

Сценарій «Бізнес-Бізнес». Сценарій взаємодії бізнес-процесів у процесі функціонування Web-орієнтованого програмного забезпечення, який називають «Бізнес-Бізнес», подано на рис. 2.67.

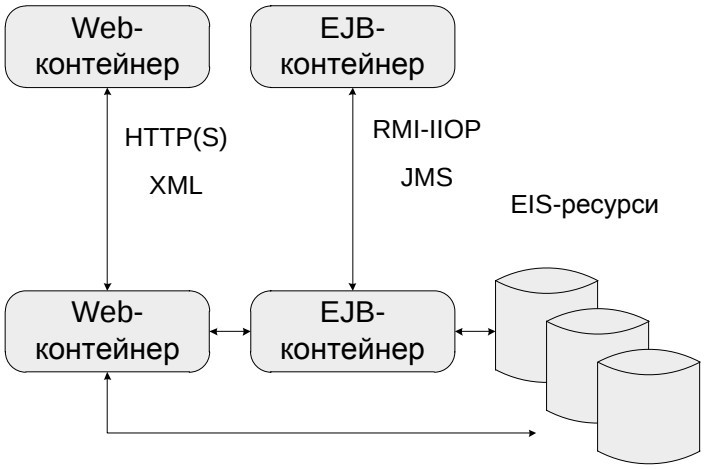


Рис. 2.67. Сценарій Бізнес-Бізнес

Цей сценарій акцентує увагу на міжрівневій взаємодії між Web- і EJB-контейнерами. Модель програмування J2EE пропонує використовувати XML-дані як головний засіб комунікації між Web-контейнерами, що зручно у разі розробки та розгортання комерційних рішень, в основу яких покладено Web. Міжрівневі комунікації між EJB-контейнерами в даний час найкраще рішення для Internet-середовища.

2.9. Висновки

1. Для створення відкритої розподіленої системи необхідне використання загальноприйнятих мов опису інтерфейсу програмної компоненти.

2. На сьогодні наявна низка апробованих на практиці стандартів для передачі даних у гетерогенних розподілених системах: XML, XSD, SOAP і WSDL.

3. Використання стандартів дозволяє створювати розподілені системи, не прив'язані жорстко до засобу розробки програм або транспортного протоколу.

4. Відкритий характер специфікації SOAP допускає як реалізацію деякої додаткової функціональності, що застосовується у проміжному середовищі, так і прийняття нових стандартів і розширень, які використовують заголовки SOAP.

5. Разом з тим, це може зумовити певні труднощі у разі взаємодії, заснованих на WSDL і SOAP програмних компонентів різних розробників.

6. Платформи розглянуто з погляду побудови на них складних гетерогенних Web-систем, тому деякі з популярних технологій не наведено в детальному огляді через неможливість або недоцільність їх використання як базової платформи. З розглянутого можна виокремити такі основні підходи до побудови архітектури серверних прикладних програм:

окреме виконання запитів, коли під час кожного запиту динамічного вмісту запускається окрема програма для обробки запитів, яка генерує вміст, що передається клієнтові. Цей підхід використовують у класичних CGI-скриптах;

накопичення виконуваних процесів, підхід аналогічний попередньому, але, якщо запит виконується повторно, то заново програма не запускається, а обробка передається існуючому процесу. Такий підхід застосовують у технологіях Java Servlets, Fast CGI;

шаблони сторінок, коли під час запиту шаблони заповнюють динамічним вмістом, який інтерпретується мовою сценаріїв, але не завжди створений у обчислювальному процесі. Підхід застосовують у технологіях ASP, JSP, PHP;

розширення Web-сервера, коли Web-сервер використовує особливі розширення для обробки динамічного змісту, специфічні для Web-сервера. Цей підхід використовують в IS API, NSAPI, mod_perl.

7. Rich Internet Application – це прикладна програма, доступна через Internet, збагачена функціональністю традиційних настільних прикладних програм, не підтримується браузерами безпосередньо. У RIA значна частина функціонала винесена у клієнтську частину, тобто виконується на стороні клієнта.

8. Основні механізми дескрипторів розгортання Web-орієнтованих прикладних програм на платформі J2EE реалізують такі компоненти платформи як EJB-контейнер та Web-контейнер. Наявні безліч способів, якими можливо реалізовувати специфічний набір бізнес-вимог, логічну модель і описувати рішення, що задовольняють поставленим вимогам. Ураховуючи ці вимоги, складають різноманітні сценарії функціонування прикладних програм, серед яких можна назвати сценарій трирівневої Web-прикладної програми, який дозволяє не застосовувати EJB-контейнер, тобто використовувати менше ресурсів та витрачати менше часу і процесорної пам’яті.

2.10. Запитання для самоконтролю

1. У чому полягають відмінності між мультипроцесорами та мультикомп’ютерами?
2. Коротко охарактеризуйте гомогенні та гетерогенні комп’ютерні розподілені системи.
3. На які категорії поділяють операційні системи для розподілених комп’ютерів? Коротко охарактеризуйте кожну з них.
4. Що являє собою проміжне середовище?
5. Назвіть основні служби платформи розподілу.
6. Дайте визначення поняття «розподілене середовище».

7. Назвіть основні компоненти, які беруть участь у процесі взаємодії, та опишіть процес взаємодії в розподіленому середовищі.

8. На які логічні рівні можна поділити прикладні програми. Охарактеризуйте кожний з них?

9. Назвіть і коротко охарактеризуйте варіанти архітектури клієнт-сервер.

10. Дайте визначення програмної компоненти.

11. Які умови необхідні для повного формального опису взаємодії двох компонент розподіленої системи?

12. Що являє собою інтерфейс компоненти розподіленої системи?

13. Що є основою мережної взаємодії віддалених систем?

14. Назвіть сервіси, які мають бути застосованими у проміжному середовищі для взаємодії компонент розподіленої системи?

15. Назвіть і коротко охарактеризуйте наявні концепції взаємодії програмних компонент.

16. Назвіть можливі варіанти віддаленого виклику процедур.

17. Дайте визначення понять «маршалізація», «серіалізація», «десеріалізація».

18. Що собою являють віддалені об’єкти? Для чого їх використовують?

19. Назвіть моделі використання віддалених об’єктів.

20. Коротко охарактеризуйте модель єдиного виклику.

21. Коротко охарактеризуйте модель єдиного екземпляра.

22. Коротко охарактеризуйте модель активації об’єктів за запитом клієнта.

23. Назвіть і охарактеризуйте підходи до обробки розподілених подій.

24. Що таке розподілена транзакція?

25. Які якості притаманні розподіленій транзакції?

26. Підтримку яких функцій має забезпечувати проміжне середовище для гарантування безпеки розподіленої системи?

27. Назвіть основні переваги та недоліки мови XML.

28. Яку роль відіграє SOAP у процесі побудови розподілених гетерогенних систем?

29. У чому полягає суть процесів серіалізації та десеріалізації об’єктів?

30. Які є основні підходи до побудови прикладних програм?

31. Які базові технології використовують для Web-орієнтованих прикладних програм?

32. Яким чином реалізується розподіл платформ за схемою обробки запитів?

33. Що таке RIA-прикладна програма?

34. Чим RIA відрізняється від архітектур товстого і тонкого клієнта?

35. Які технології RIA найчастіше використовують нині?

36. Яку роль відіграє EJB-контейнер у розробці Web-орієнтованих прикладних програм?

37. Які служби реалізує Web-контейнер?

38. Опишіть механізм виконання та переваги багаторівневого сценарію програми.

3. ЗВ'ЯЗОК

Зв'язок між процесами – це спосіб взаємодії й обміну даними між процесами. Сучасні розподілені системи часто містять тисячі процесів, розкиданих ненадійною мережею, наприклад Internet. Щоб спростити розробку масштабних прикладних систем, необхідно замінити найпростіші засоби взаємодії в телекомунікаційних мережах.

3.1. Рівні протоколів

Еталонна модель взаємодії відкритих систем (Open Systems Interconnection Reference Model) OSI/ISO спрощує роботу з безліччю рівнів і понять, використовуваних у передачі даних, та чітко визначає сім рівнів, дає їм стандартні імена і вказує, який рівень за що відповідає. Модель OSI розроблено для того, щоб надати відкритим системам можливість взаємодіяти.

Відкритою називають систему, здатну взаємодіяти з будь-якою іншою відкритою системою за стандартними правилами, що визначають формат, вміст і зміст повідомлень, які відправляються і приймаються.

Ці правила зафіксовані в чинних стандартах і їх називають **протоколами** (protocols). Усі протоколи поділяють на два основні типи. Перший тип – протоколи *зі встановленням з'єднання (connection-oriented),* у яких перед початком обміну даними відправник і одержувач мають встановити з'єднання і, можливо, домовитися про те, який протокол вони використовуватимуть. Після завершення обміну вони мають розірвати з'єднання. Системою зі встановленням з'єднання є, наприклад, телефон. Другий тип – використання протоколів *без установлення з'єднання (connectionless),* у такому разі підготовки не потрібно. Відправник надсилає перше повідомлення, як тільки він готовий це зробити.

У моделі OSI взаємодію поділяють на сім рівнів, як показано на рис. 3.1. Кожен рівень відповідає за один специфічний аспект взаємодії, надає інтерфейс для роботи з вищим рівнем. Інтерфейс містить набір операцій, які ра-

зом визначають інтерфейс, що надається тим рівнем, яким користується поточний рівень.

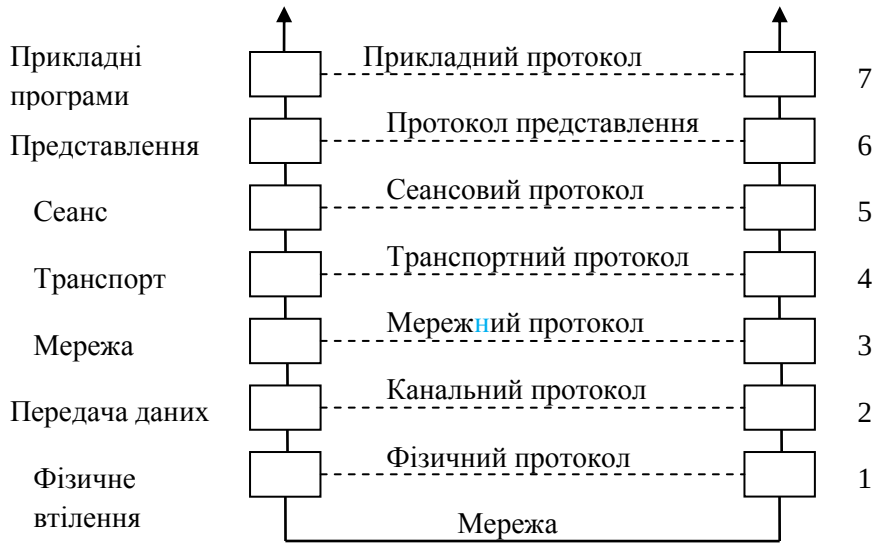


Рис. 3.1. Рівні, інтерфейси і протоколи OSI

Приклад. Коли процес *A* на машині *1* хоче поспілкуватися з процесом *B* на машині *2*, він створює повідомлення й надсилає його прикладному рівню своєї машини. Програмне забезпечення прикладного рівня додає в початок повідомлення свій *заголовок (header)* і передає отримане повідомлення через інтерфейс із рівня *7* на рівень *6*. Рівень представлення (6), у свою чергу, додає в початок повідомлення свій заголовок і передає результат вниз, на сеансовий рівень (5) і т. д. Деякі рівні додають не лише заголовок у початок, але і завершення в кінець. Коли повідомлення дійде до фізичного рівня, останній виконає його реальну передачу, як це показано на рис. 3.2.

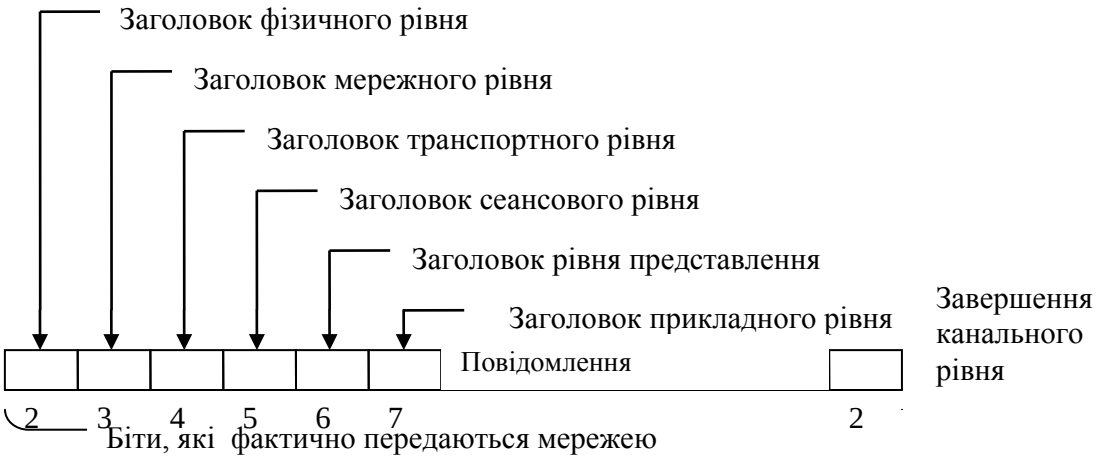


Рис. 3.2. Формат повідомлення, переданого мережею

Коли повідомлення надходить на машину *2*, воно передається вгору, при цьому на кожному рівні зчитується й перевіряється відповідний заголовок. Зрештою повідомлення досягає одержувача, процеса *B*, що може відповісти на нього, у процесі надання відповіді одержувачем усе повторюється у зворотному напрямку. Інформацію із заголовка рівня *n* використає протокол рівня *n*.

3.1.1. Низькорівневі протоколи

Фізичний рівень. Фізичний рівень відповідає за передачу нулів й одиниць, а його протоколи - за стандартизацію електричних, механічних і сигнальних інтерфейсів, щоб, якщо одна машина надсилає нуль, то друга прийняла його як нуль, а не як один: ADSL (Asymmetric Subscriber Line), ISDN (Integrated Services Digital Network), SONET (Synchronous Optical Networking), стандартні модемні протоколи (протоколи серії ITU V), що використовуються в з'єднаннях між аналоговими модемами телефонною лінією тощо.

Канальний рівень. Фізичний рівень тільки пересилає біти. Поки немає помилок, все добре, однак у реальних мережах з'являються помилки, їх потрібно знаходити й виправляти. Це і є основним завданням канального рівня, оскільки він групує біти в модулі, які зазвичай називають **кадрами** (frames), і стежить за тим, щоб кожен кадр було передано правильно за рахунок розміщення спеціальної бітової маски в початок і кінець кожного кадру для їх маркування, а також за рахунок обчислення **контрольної суми** *(checksum)*, тобто підсумовування всіх байтів кадру. Канальний рівень додає контрольну суму до кадру. Коли кадр приймається, приймач повторно обчислює контрольну суму даних і порівнює результат з контрольною сумою, що надійшла разом з кадром. Якщо вони збігаються, то кадр уважається правильним і приймається; якщо різні, то одержувач просить відправника повторно надіслати цей кадр. Кадри послідовно нумеруються із зазначенням номерів у заголовку, таким чином, що всі розуміють, де який кадр.

Приклад. Протоколи: ARCNET (Attached Resource Computer NETwork), CDP (Cisco Discovery Protocol), DCAP (Data Link Switching Client Access Protocol), DTP (Dynamic Trunking Protocol), L2FP (Layer 2 Forwarding Protocol), L2TP (Layer 2 Tunnelling Protocol) тощо.

Мережний рівень. Повідомлення, яке надсилається від відправника до одержувача, має пройти безліч мережних сегментів, на кожному з яких обирається вихідний напрям. Вибір найкращого шляху називають *маршрутизацією (routing)* і це є основним завданням мережного рівня.

Нині найбільш поширеним мережним протоколом, що не потребує встановлення з'єднання, є *протокол Internet (Internet protocol, IP)*, який входить до стеку протоколів Internet. Для іменування повідомлення на мережному рівні використовують термін «**пакет»** (packet). IP-пакет може бути надісланий без попередньої підготовки, маршрут кожного з IP-пакетів до місця призначення обирається незалежно від інших пакетів. Жодний внутрішній шлях не обирається заздалегідь і не запам'ятовується.

Приклад. Протоколом із з'єднанням є *віртуальний канал (virtual channel)* на основі мереж ATM (Asynchronous Transfer Mode), мережний протокол без установлення з'єднання CLNP (Connection Less Network Protocol).

Віртуальний канал в ATM – це непряме з'єднання, встановлене від джерела до приймача, яке може проходити через кілька проміжних ATM-комутаторів.

Протоколи *без установлення з'єднання:* протокол IPv4(Internet Protocol version 4), IPv6 (Internet Protocol version 6), IGRP (Interior Gateway Routing Protocol), IPSec (Internet Protocol Security) тощо.

3.1.2. Транспортні протоколи

Транспортний рівень – це верхня частина того, що називають базовим стеком мережних протоколів, оскільки в ньому реалізовано всі служби, які необхідні для побудови мережних прикладних програм і які не ввійшли в інтерфейс мережного рівня.

Після одержання повідомлення з прикладного рівня транспортний рівень розбиває його для успішної передачі на достатньо дрібні частини, призначає їм послідовні номери й пересилає їх. Взаємодія на рівні заголовка транспортного рівня зводиться до обговорення того, який пакет був посланий, який прийнятий, скільки місця є в адресата для прийому подальших повідомлень, що варто послати повторно тощо.

Приклад. Транспортний протокол для Internet називають *протоколом керування передачею (Transmission Control Protocol, TCP)*. Комбінація TCP/IP зараз є стандартом де-

факто у разі мережної взаємодії. Комплект протоколів Internet також містить транспортний протокол *UDP (Universal Datagram Protocol – універсальний протокол датаграм)*, який не потребує з'єднання та фактично є протоколом IP з деякими незначними доповненнями.

Офіційний транспортний протокол ISO має п'ять різновидів – від TP0 до TP4. Відмінності полягають у процесах обробки помилок і можливості працювати з декількома транспортними з'єднаннями на основі одного з'єднання низького рівня (особливо X.25). Вибір того, який з них використати, залежить від властивостей протоколу, який лежить нижче мережного рівня. Жоден з них не має перевантажуватися.

3.1.3. Протоколи верхнього рівня

Сеансові протоколи і протоколи представлення. Сеансовий рівень фактично є розширеною версією транспортного рівня, оскільки забезпечує керування діалогом, відслідковуючи й запам'ятовуючи, яка сторона говорить у цей момент, і надає інформацію синхронізації. Ця інформація потрібна, щоб користувачі створювали контрольні точки у разі довгих сеансів передачі даних та повідомлення про збій у процесі такого сеансу. Контрольні точки необхідні для того, щоб зробити відкат лише до останньої контрольної точки, а не проходити весь шлях спочатку. На практиці сеансовий рівень використовується деякими прикладними програмами, підтримується рідко і навіть не входить до комплекту протоколів Internet.

Рівень представлення відповідає за зміст переданих бітів. Більшість повідомлень містять не випадкові послідовності бітів, а структуровану інформацію (прізвища, адреси, грошові суми тощо). На рівні представлення можна визначити записи, які містять подібні поля, і вимагати у відправника повідомляти одержувача, що повідомлення містить окремі записи відповідного формату.

Прикладні протоколи. Прикладний рівень моделі OSI містить набір стандартного мережного прикладного програмного забезпечення, наприклад для роботи з електронною поштою, для передачі файлів й емуляції термінала. На сьогодні він став місцем розташування всього прикладного програмного забезпечення і протоколів, які не вдалося розмістити на жодному з нижчих рівнів. З погляду еталонної моделі OSI всі розподілені системи є просто розподіленим між обчислювальними вузлами прикладним програмним забезпеченням.

Приклад. Популярний в Internet *протокол передачі файлів (File Transfer Protocol, FTP)* визначає передачу файлів між клієнтською машиною й сервером. Цей протокол не слід плутати з програмою *ftp*, що являє собою прикладну програму для передачі файлів і збігається (але не повністю) з реалізацією протоколу FTP для Internet.

Іншим прикладом спеціального прикладного протоколу може бути *протокол передачі гіпертексту (Hypertext Transfer Protocol, HTTP)*, розроблений для віддаленого керування й завантаження web-сторінок, реалізований у таких прикладних програмах, як web-браузери й web-сервери.

Протоколи проміжного рівня. До **проміжного рівня** належать прикладні програми, які розміщують логічно на прикладному рівні, але використовують низку протоколів загального призначення, що дає їм право на власний рівень, який не залежить від інших та охоплює більше спеціалізованих прикладних програм. Можна відокремити *високорівневі протоколи взаємодії* від *протоколів для надання різних служб* проміжного рівня.

Приклад. Як приклад розглянемо групу *протоколів розподіленого підтвердження (commit),* які розробляють так, щоб окрема група процесів або всі процеси пройшли через певну операцію, або операцію не було застосовано до жодного з них. Це явище відоме також як атомарність, яку широко використовують у процесі виконання транзакцій. Не тільки транзакції, але й інші прикладні програми, особливо які мають бути стійкими до збоїв, також можуть мати потребу в використанні протоколів розподіленого підтвердження. До таких протоколів належать такі: *протокол SOAP (Simple Object Access Protocol)*, група *протоколів XMPP (Extensible Messaging and Presence Protocol), протокол UMSP (Unifired Memory Space Protocol)* та інші.

Комунікаційні протоколи проміжного рівня підтримують високорівневі комунікаційні служби, з яких розглянемо лише чотири високорівневі комунікаційні служби проміжного рівня: віддалений виклик процедур, віддалене звертання до об’єктів, черги повідомлень і потоки даних.

Такий підхід до поділу на рівні зумовлює дещо змінену еталонну модель взаємодії (рис. 3.3). Порівняно з моделлю OSI сеансовий рівень і рівень представлення замінено одним проміжним рівнем, що містить протоколи, які не залежать від прикладних програм і їх не можна помістити на нижчі рівні. Транспортні служби також можуть бути подані у вигляді служб проміжного рівня, не потребуючи навіть модифікації. Цей підхід аналогічний перенесенню UDP на транспортний рівень.

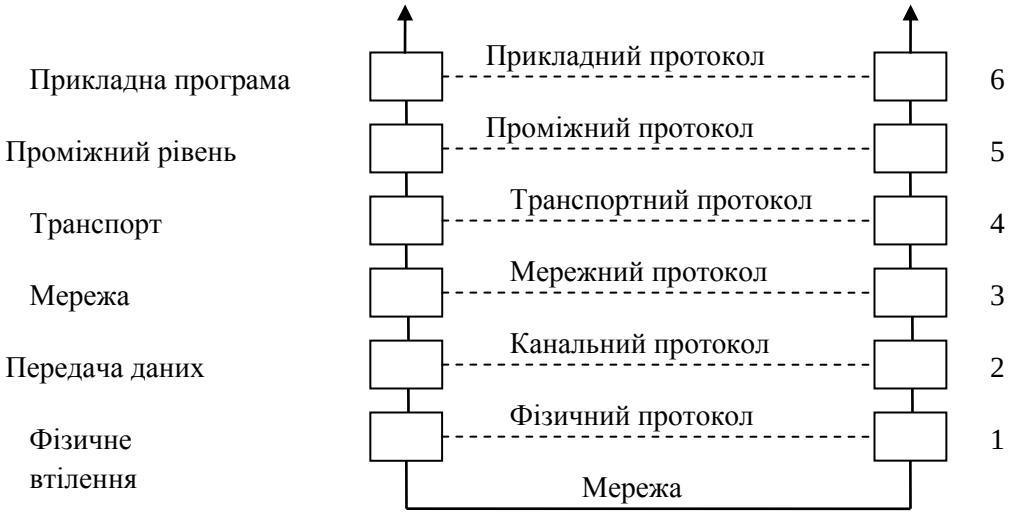


Рис. 3.3. Змінена еталонна модель мережної взаємодії

3.2. Віддалений виклик процедур

Основою більшості розподілених систем є явний обмін повідомленнями між процесами, однак процедури *send* та *receive* не приховують взаємодії, необхідної для забезпечення прозорості доступу. Було запропоновано дозволити програмам викликати процедури, які перебувають на інших машинах. Коли процес, запущений на машині *A*, викликає процедуру з машини *B*, то процес, який викликається на машині *A* припиняється, а виконання викликаної процедури відбувається на машині *B*. Інформація може бути передана від процесу, який викликає, до процедури, яка викликається, через параметри й повернута процесу у вигляді результату виконання процедури. Цей метод називають **віддаленим викликом процедур** (Remote Procedure Call, RPC*).*

Ідея **виклику віддалених процедур** полягає в розширенні добре відомого і зрозумілого механізму підміни процесу керування обчисленнями й даними, які містяться в середині програми, що виконується на одній машині, на процес, який виконується через мережу. Засоби віддаленого виклику процедур призначені для полегшення організації розподілених обчислень і створення розподілених клієнт-серверних інформаційних систем. Найбільша ефективність використання RPC досягається в тих прикладних програмах, у яких наявний інтерактивний зв’язок між віддаленими компонентами з невеликою

тривалістю відповідей і порівняно малою кількістю переданих даних. Такі прикладні програми називають RPC-орієнтованими.

Характерні риси виклику локальних процедур такі: асиметричність, тобто одна зі сторін, які взаємодіють, є ініціатором; синхронність, тобто виконання процедури, яка викликає віддалену процедуру, припиняється з моменту виклику нею запиту й відновлюється лише після повернення результатів запиту з викликаної процедури.

Реалізація віддалених викликів значно складніша від реалізації викликів локальних процедур. Можна назвати проблеми й завдання, які необхідно вирішити у процесі реалізації RPC: оскільки процедура, яка викликає, й процедура, яку викликають, виконуються на різних машинах, то вони мають різні адресні простори, що створює проблеми під час передачі параметрів і результатів, зокрема якщо машини керовані різними операційними системами або мають різну архітектуру (наприклад, використовується прямий або зворотний порядок байтів). Через те, що RPC не може розраховувати на поділювану пам'ять, параметри RPC не мають містити вказівників на чарунки нестекової пам'яті, а значення параметрів – копіюватися з одного комп'ютера на другий. Для копіювання параметрів процедури й результату її виконання через мережу виконується їх серіалізація. Під **серіалізацією** розуміють процес переведення будь-якої структури даних у послідовність бітів. Оберненою до серіалізації є операція **десеріалізації** – відновлення початкового стану структури даних з бітової послідовності. Серіалізацію використовують для передачі об'єктів мережею й для збереження їх у файли.

Приклад. Розглянемо, як використовують серіалізацію для розподіленого програмного забезпечення, різні частини якого мають обмінюватися даними зі складною структурою. У такому разі для типів даних, які передбачається передавати, використовується код, який здійснює серіалізацію й десеріалізацію. Об'єкт заповнюється потрібними даними, потім викликається код серіалізації, за допомогою якого створюється XML-документ. Результат серіалізації передається приймальній стороні, наприклад електронною поштою або HTTP. Програмне забезпечення одержувача створює об'єкт того ж типу й викликає код десеріалізації, у результаті чого одержується об'єкт із тими ж даними, які були в об'єкта програмного забезпечення відправника. За такою схемою працює, зокрема, серіалізація об'єктів з використанням протоколу SOAP у Microsoft .NET.

На відміну від локального, віддалений виклик процедур обов'язково використовує транспортний рівень мережної архітектури (наприклад, TCP), однак цей факт залишається прихованим від розробника програмного забезпечення.

Виконання програми, яка викликає віддалену процедуру, і викликуваної локальної процедури в одній машині реалізується у межах єдиного процесу, але в реалізації RPC беруть участь як мінімум два процеси – по одному в кожній машині. У разі, якщо один з них завершиться аварійно, можливі такі ситуації: у разі аварії процедури, яка викликає віддалені процедури, віддалено викликані процедури втратять процедури верхнього рівня, а у разі аварійного завершення віддалених процедур, процедури верхнього рівня втратять підпорядковані процедури.

Виникає низка проблем, зумовлених неоднорідністю мов програмування й операційних середовищ: структури даних і структури виклику процедур, підтримувані в будь-якій мові програмування, не підтримуються так само в усіх інших мовах, тобто наявна проблема сумісності, яку досі не розв'язано ні за рахунок введення одного загальноприйнятого стандарту, ні за рахунок реалізації декількох конкуруючих стандартів на всіх архітектурах і в усіх мовах.

Приклад. Припустімо, програма має зчитати деякі дані з файлу. Для читання з файлу необхідних даних програміст поміщає в код виклик *read*. У традиційній (однопроцесорній) системі процедура *read* витягається компонувальником з бібліотеки й вставляється в об'єктний код програми. Навіть якщо *read* – це системний виклик, він відпрацьовується аналогічно, через розміщення параметрів у стек. RPC організує свою прозорість у такий же спосіб. Якщо *read* є віддаленою процедурою (тобто буде виконуватися на машині файлового сервера), то у бібліотеці розміщується спеціальна версія *read*, яку називають *клієнтською заглушкою (client stub).* Як і оригінальна функція, вона також викликається відповідно до розглянутої послідовності і викликає локальну операційну систему, але, на відміну від оригінальної функції, клієнтська заглушка не запитує даних в операційної системи, а упаковує параметри в повідомлення й викликом процедури *send* вимагає переслати це повідомлення на сервер, як показано на рис. 3.4. Після виклику процедури *send* клієнтська заглушка викликає процедуру *receive*, блокуючись до отримання відповіді.

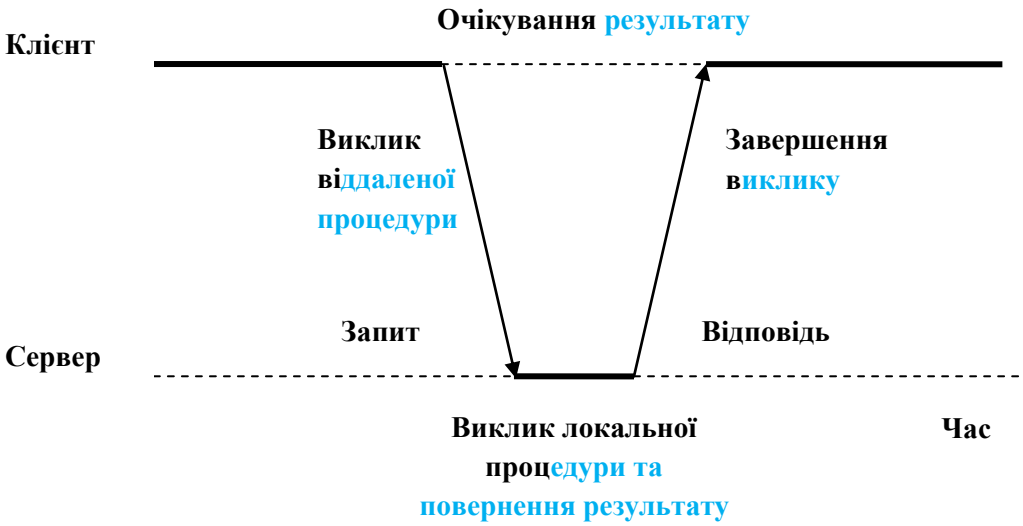


Рис. 3.4. Схема RPC між програмами клієнта й сервера

Коли повідомлення надходить на сервер, операційна система сервера передає його серверній заглушці (*serverstub*), яка є еквівалентною клієнтській, але працює на стороні сервера.

Підведемо підсумки. У разі віддаленого виклику процедур відбуваються такі дії:

1. Процедура клієнта звичайним способом викликає клієнтську заглушку.
2. Клієнтська заглушка створює повідомлення й викликає локальну операційну систему.
3. Операційна система клієнта пересилає повідомлення віддаленій операційній системі.
4. Віддалена операційна система передає повідомлення серверній заглушці.
5. Серверна заглушка вилучає з повідомлення параметри й викликає сервер.
6. Сервер виконує виклик і повертає результати заглушці.
7. Серверна заглушка запаковує результати в повідомлення й викликає свою локальну операційну систему.
8. Операційна система сервера пересилає повідомлення операційній системі клієнта.
9. Операційна система клієнта приймає повідомлення й передає його клієнтській заглушці.
10. Заглушка вилучає результати з повідомлення й передає їх клієнтові.

Ефекти взаємодії в мережі за допомогою цих кроків полягають у тому, що клієнтська заглушка перетворює локальний виклик процедури клієнта в локальний виклик процедури сервера, причому ні клієнт, ні сервер нічого не знають про проміжні дії.

3.3. Звертання до віддалених об'єктів

Розглянемо, як реалізує RPC звертання до віддалених об'єктів, і як подібний підхід підвищує прозорість мережної взаємодії (розподілу програмного забезпечення між вузлами мережі) порівняно з викликами віддалених процедур RPC.

3.3.1. Розподілені об'єкти

Ключова особливість об'єкта полягає в тому, що він інкапсулює стани й методи (*methods*): **стани** – дані, які інкапсулює об'єкт; **методи** – операції над даними, які також інкапсулює об'єкт. Доступ до методів можна одержати через **інтерфейс.** Важливо зрозуміти, що єдино правильним способом доступу або маніпулювання станом об'єкта є використання методів, доступ до яких здійснюється через інтерфейс цього об'єкта, який може реалізовувати декілька інтерфейсів. Для такого опису інтерфейсу може бути кілька об'єктів, які надають його реалізацію.

Розподіл на інтерфейси й об'єкти, які реалізують інтерфейси, є дуже важливим для розподілених систем, оскільки чіткий поділ дозволяє розміщувати інтерфейс на одній машині за умови, що сам об'єкт перебуває на другій.

Структуру розподіленого об'єкта (distributed object) показано на рис. 3.5. Коли клієнт виконує прив'язку до розподіленого об'єкта, в адресний простір клієнта завантажується реалізація інтерфейсу об'єкта, яку називають **замісником** (*proxy*). Замісник клієнта є аналогічним клієнтській заглушці в системах RPC. Єдине, для чого він призначений, – виконувати *маршалінг* параметрів у повідомленнях у разі звертання до методів і *демаршалінг* даних із повідомлень відповіді, які містять результати звертання до методів, передаючи їх клієнтові. Самі об'єкти перебувають на сервері й надають необхідні клієнтсь-

кій машині інтерфейси. Вхідний запит на звертання до методу спочатку потрапляє на серверну заглушку.

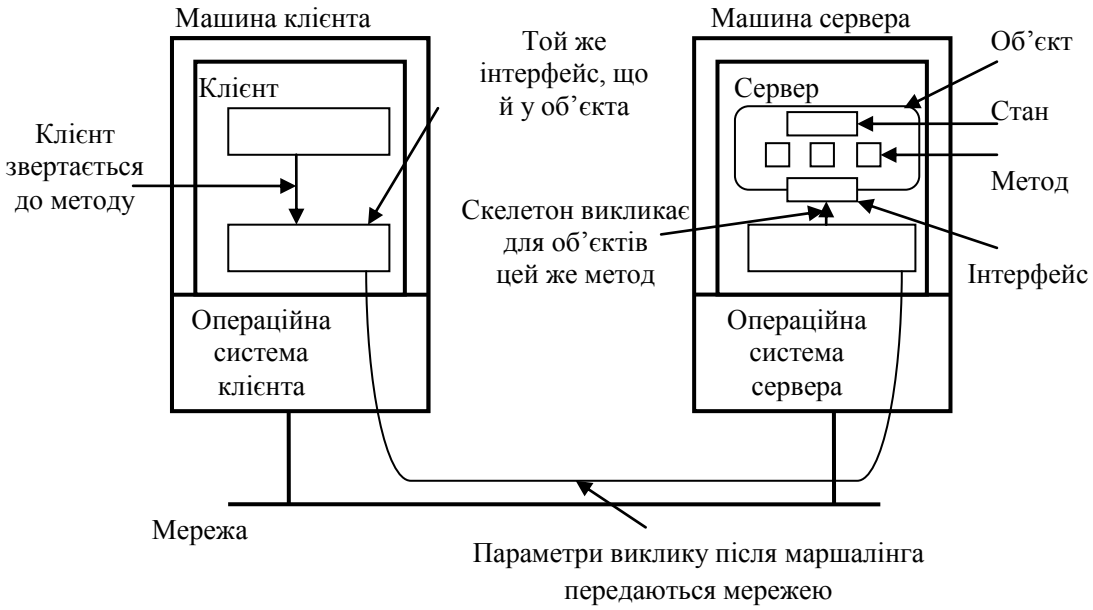


Рис. 3.5. Узагальнена організація віддалених об'єктів з використанням замісника клієнта

Скелетон – серверна заглушка, яка перетворить звертання клієнта у правильне звертання до методу через інтерфейс об'єкта, який перебуває на сервері. **Серверна заглушка** також відповідає за маршалінг параметрів у повідомленнях-відповідях і пересилання їх замісникові клієнта.

3.3.2. Прив'язка клієнта до об'єкта

Відмінність між традиційними системами RPC і системами, що підтримують розподілені об'єкти, полягає в тому, що останні зазвичай надають посилання на об'єкти, унікальні в межах системи. Коли процес зберігає посилання на об'єкт, перед звертанням до кожного з методів об'єкта процес має передусім виконати прив'язку до цього об'єкта. Результатом прив'язки буде замісник, який розташовується в адресному просторі процесу й реалізує інтерфейс із методами, до яких звертається процес. Це означає, що клієнтові надається простий механізм, який дозволяє напряму запитувати методи, використовуючи тільки посилання на об'єкт. У разі неявної прив'язки клієнт

прозоро зв'язується з об'єктом у момент дозволу на виконання посилання й одержання цього об'єкта насправді.

Явна прив'язка *(explicit binding)* полягає у тому, що клієнт має перед звертанням до методу викликати спеціальну функцію для прив'язки до об'єкта.

3.3.3. Статичне й динамічне віддалене звертання до методів

Віддалене звертання до методів (RMI) – звертання клієнта через замісника до методів об'єкта, тобто виклик спеціальної функції (замісника) для звертання до методів об'єкта.

Стандартний спосіб підтримки RMI – описати інтерфейси об'єктів мовою визначення інтерфейсів, як і у RPC. Окрім того, можна використати об'єктну мову, наприклад Java, що забезпечує автоматичне генерування заглушок. Такий підхід до застосування попередньо визначених описів інтерфейсів називають **статичним звертанням** *(static invocation).*

Статичне звертання *(static invocation)* вимагає, щоб інтерфейси об'єкта під час розробки клієнтської прикладної програми були відомі, а також передбачає, що зі зміною інтерфейсу клієнтську прикладну програму перед використанням нових інтерфейсів буде перекомпільовано.

Альтернативою є звертання до методів, яке може здійснюватися більш динамічно, зокрема, іноді зручніше визначити параметри звертання до методу під час виконання. Цей процес відомий під назвою **«динамічного звертання»** *(dynamic invocation)*, тобто звертання, під час виконання якого прикладна програма обирає, який метод віддаленого об'єкта буде викликаний.

Динамічне звертання зазвичай записують таким чином: *invoke (object, method. input_parameters. output_parameters),* де *object* – ідентифікує розподілений об'єкт; *method* – параметр, який точно задає метод, який викликається; *input_parameters* – структура даних, у якій утримуються значення вхідних параметрів методу; *output_parameters* – структура даних, у якій зберігаються значення, які повертаються.

Приклад. Розглянемо додавання цілого числа *int* до об'єкта *fobject* файлу. Для виконання цієї дії об'єкт надає метод *append*. У такому разі статичне звертання матиме

вигляд *fobject.append (int),* а динамічне – *invoke(fobject, id(append). int),* де операція *id(append)* повертає ідентифікатор методу *append*.

Інша область застосування динамічних звертань – *служби пакетної обробки*, для яких запити на звертання можуть оброблятися протягом часу, поки звертання очікує виконання.

3.3.4. Передача параметрів

Розглянемо ситуацію, коли всі об'єкти в системі є розподіленими, тобто доступними з віддалених машин. У цьому разі під час звертань до методів потрібно постійно використовувати посилання на об'єкти, як на параметри. Посилання передаються за значенням і копіюються з однієї машини на другу. У такому разі це посилання копіюється й передається як параметр-значення лише тоді, коли воно вказує на віддалений об'єкт. Саме в цьому разі відбувається передача об'єкта за посиланням. Однак, якщо посилання вказує на локальний об'єкт, тобто об'єкт в адресному просторі клієнта, то об'єкт, на який указує посилання, повністю копіюється й у процесі звертання передається клієнтові. У такому разі об'єкт передається за значенням.

Приклад. Ці два способи реалізації звертань показано на рис. 3.6, на якому зображені клієнтська програма, яка виконується на машині *A*, і програма-сервер, яка виконується на машині *C*. Клієнт володіє посиланням на локальний об'єкт *01,* який використовується як параметр під час виклику серверної програми на машині *C,* а також посиланням на віддалений об'єкт *02*, який міститься на машині *B* і також використовується як параметр. Під час виклику сервера на машину *C* передається копія всього об'єкта *01* і копія посилання на об'єкт *02*.

Додатковий ефект звертання до методів з використанням як параметра посилання на об'єкт полягає в можливості копіювати об'єкти, які приховати неможливо, а тому потрібно явно вказувати на відмінності між локальними й розподіленими об'єктами.

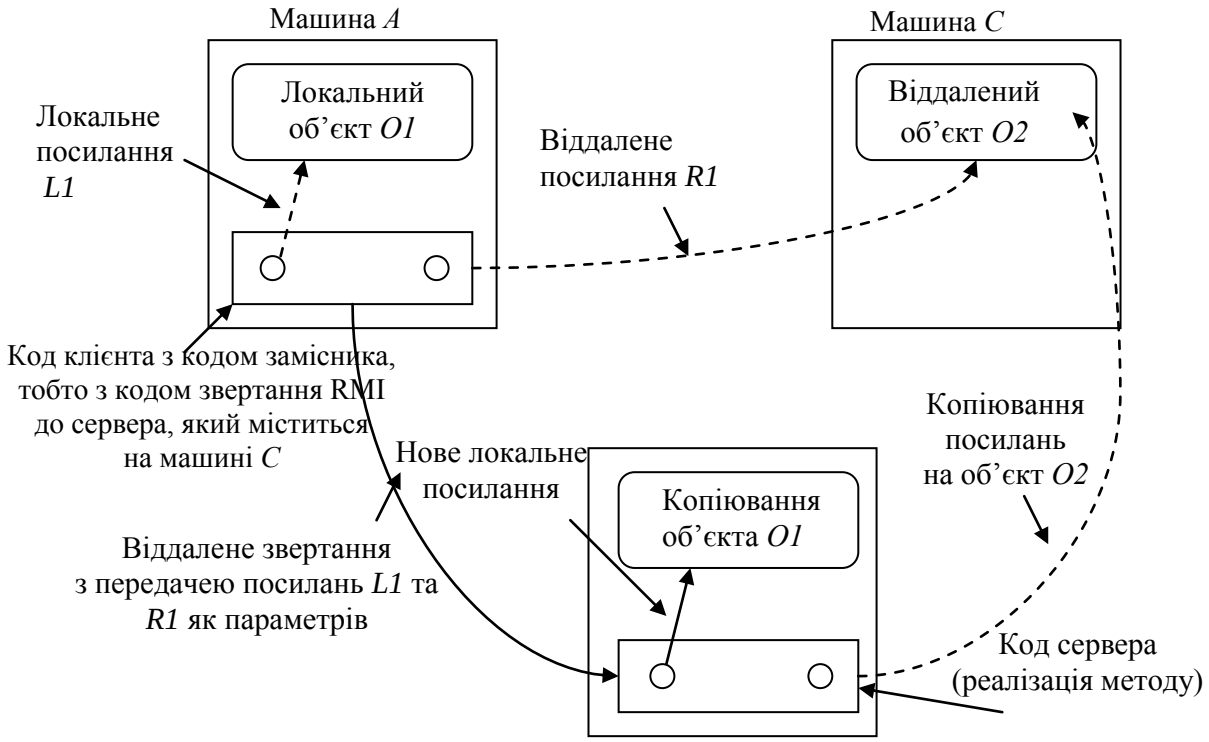


Рис. 3.6. Передача об'єкта за посиланням і за значенням

3.4 Зв'язок на основі потоків даних

Розглянемо, які засоби можуть використовуватися розподіленими системами для роботи з інформацією, критичною до часових характеристик передачі, зокрема з відео- та аудіопотоками.

3.4.1 Підтримка безперервних середовищ

Підтримка обміну критичною до часових характеристик передачі інформацією часто зводиться до підтримки **безперервних середовищ.** Під **середовищем** розуміють те фізичне середовище, що несе інформацію. Найважливіша характеристика середовища – спосіб представлення інформації.

Безперервне середовище представлення (continuous representation media) – це часові співвідношення між різними елементами даних, які лежать в основі коректної інтерпретації змісту даних.

Приклад. Одержання звуку під час відтворення аудіопотоку.

Дискретне середовище представлення (discrete representation media) характеризують тим, що часові співвідношення між елементами даних не відіграють суттєвої ролі у правильній інтерпретації даних.

Приклад. Типовими прикладами дискретного середовища є представлення тексту і статичні зображення, а також об'єктний код і файли, які використовуються.

Потік даних. Для обміну критичною до часу передачі інформацією розподілені системи зазвичай надають підтримку потоків даних (data streams або просто streams), послідовності елементів даних.

Потоки даних використовують як для дискретного, так і для безперервного середовища представлення. Для підтримки часових характеристик часто доводиться обирати між різними режимами передачі.

Асинхронний режим передачі (asynchronous transmission mode) полягає в тому, що елементи даних передаються в потік один за одним, але на їх подальшу передачу жодних обмежень за часовими характеристиками не вводиться. Такий режим використовується для дискретних потоків даних.

Синхронний режим передачі (synchronous transmission mode) полягає в тому, що для кожного елемента в потоці даних визначається максимальна затримка наскрізної передачі. Якщо елемент даних був переданий значно швидше від максимально допустимої затримки, то це не впливає на процес його передачі, а якщо затримка більша ніж максимально допустима, то елемент буде втрачено.

Приклад. Так датчик може з певною частотою вимірювати температуру й пересилати ці дані мережею оператору. В цьому разі часові характеристики можуть становити інтерес, якщо час наскрізного проходження сигналу мережею гарантовано нижчий, ніж інтервал між вимірами, але нікому не зашкодить, якщо виміри передаватимуть значно швидше, ніж це дійсно необхідно.

Ізохронний режим передачі (isochronous transmission mode) полягає в тому, що необхідно, щоб усі елементи даних передавалися вчасно, тобто передачу даних обмежено максимальною затримкою, у цьому режимі наявні також мінімально допустимі затримки, які називають *граничним треммтінням*. Ізохронний режим передачі становить інтерес для розподілених систем мультимедіа, оскільки відіграє суттєву роль у відтворенні аудіо- і відеоінформації.

Потоки даних можуть бути простими або комплексними.

Простий потік даних (simple stream) містить лише одну послідовність даних.

Комплексний потік даних (complex stream) охоплює декілька зв'язаних простих потоків, які називають вкладеними потоками даних (substreams). Потік даних іноді може розглядатися як віртуальне з'єднання між джерелом і приймачем, а джерело або приймач може бути процесом або пристроєм.

Приклад. У процесі передачі даних через мережу процес-джерело може читати дані з аудіофайлу й пересилати їх байт за байтом мережею. Приймач може бути процесом, який вибирає байти й передає їх на локальний пристрій звуковідтворення, як зображено на рис. 3.7, *а*. Натомість, у розподілених мультимедійних системах можна реалізувати пряме з'єднання між джерелом і приймачем. Відеопотік, створюваний камерою, може прямо передаватися на дисплей, як показано на рис. 3.7, *б*.

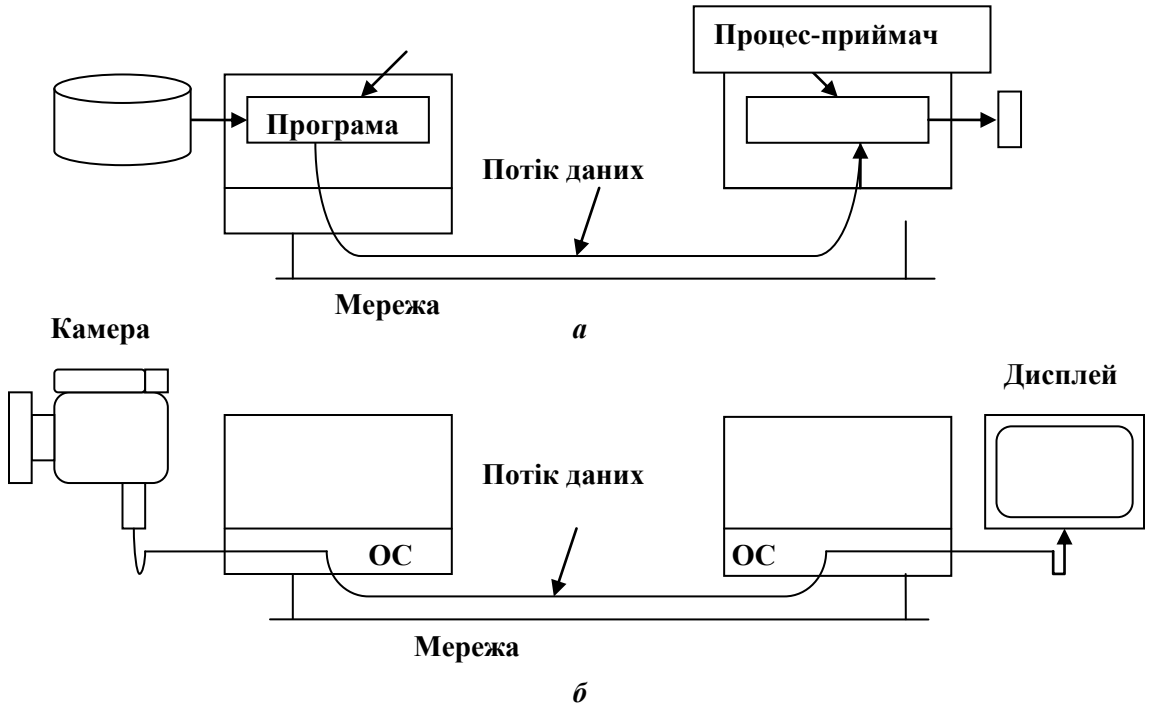


Рис. 3.7. Передача потоку даних мережею між двома процесами (*а*), між двома пристроями (*б*)

3.4.2 Потоки даних і якість обслуговування

Часові залежності й інші нефункціональні вимоги до передачі потоків даних зазвичай виражають у вигляді вимог до **якості обслуговування** (Quality of Service, QoS). Вимоги QoS для безперервних потоків даних переважно характеризують часовими діаграмами, обсягом і надійністю, які можна

подавати по-різному. Один з підходів полягає в тому, щоб надати точну специфікацію передачі (flow specification), яка містить вимоги щодо пропускної здатності, швидкості передачі, затримки тощо. Чутливість до втрат у комбінації з інтервалом втрат визначає максимально допустимий темп втрат (наприклад, один байт за хвилину).

Чутливість до групових втрат визначає, яку кількість послідовних елементів даних можна втратити.

Мінімальна затримка, яка фіксується в мережі, визначає, на скільки мережа має затримати доставку даних, щоб приймач виявив затримку. Максимальне відхилення затримки визначається максимальною межею тремтіння. Різниця між цими характеристиками особливо важлива для відео- та аудіопотоків.

Показник дотримання – число, яке показує, наскільки точно слід дотримуватися вимог до якості обслуговування.

Трафік мережі. Можна виокремити чотири найбільш загальні характеристики трафику: «вибухоподібність»; терпимість до затримок; час відповіді; ємність і пропускна здатність. Ці характеристики, з урахуванням маршрутизації, пріоритетів, з'єднань та інших ознак, визначають особливості роботи прикладного програмного забезпечення у мережі.

«Вибухоподібність» характеризує частоту надсилання трафіку користувачем, тобто чим частіше користувач надсилає свої дані в мережу, тим ця характеристика має більше значення. Користувач, який надсилає дані регулярно, в одному темпі, зводить показник «Вибухоподібність» майже до нуля. Цей показник можна визначити за відношенням максимального (пікового) значення трафіку до середнього.

Терпимість до затримок характеризує реакцію прикладного програмного забезпечення на всі види затримок у мережі. Наприклад, прикладне програмне забезпечення, яке обробляє фінансові транзакції в реальному масштабі часу, не допускає затримок, оскільки значні затримки можуть спричинити неправильну роботу такого прикладного програмного забезпечення. Прикладне програмне забезпечення розрізняють за допустимим часом затримки:

– яке працює у реальному часі (відеоконференції), для нього час затримки має бути вкрай малим;

– яке допускає затримки на кілька хвилин і навіть годин (електронна пошта, пересилання файлів).

Ємність мережі – це реальна кількість ресурсів, доступних користувачеві на певному шляху передачі даних. Пропускна здатність мережі визначається загальною кількістю даних, які можуть передаватися за одиницю часу. Ємність мережі відрізняється від пропускної здатності мережі наявністю накладних витрат, які залежать від способу використання мережі.

Можна умовно поділити трафік на три категорії, які відрізняються один від одного вимогами до затримки під час передачі.

Трафік реального часу містить аудіо- і відеоінформацію, критичну до затримок під час передачі.

Допустимі значення затримок зазвичай не перевищують 0,1 с (охоплює час на обробку пакетів кінцевою станцією), затримка має малі флуктуації, з якими пов'язаний *ефект «тремтіння»*. У разі стиснення інформації трафік цієї категорії стає дуже чутливим до помилок під час передачі, а через жорсткі вимоги до затримок у процесі передачі потоків у режимі реального часу помилки, що виникають, не можуть бути виправлені за допомогою повторного надсилання.

Трафік транзакцій. У разі передачі цього виду трафику затримки не мають перевищувати 1 с, інакше користувачі будуть змушені переривати роботу й чекати відповіді на свої повідомлення, тому що тільки після одержання відповіді вони можуть продовжити відправляти свої дані. У деяких випадках перевищення допустимого часу затримок призводить до збою робочої сесії.

Трафік даних. Затримки у разі передачі трафіку цієї категорії можуть мати фактично будь-які значення й досягати навіть декількох секунд. Для такого трафіку смуга пропускання більш важлива, ніж час затримок, оскільки збільшення пропускної здатності мережі спричиняє зменшення часу передачі. Прикладне програмне забезпечення, яке передає значні обсяги даних, розроблено здебільшого так, що захоплює всю доступну смугу пропускання мережі. Винятком є тільки програмне забезпечення потокового відео, для якого важливі й пропускна здатність, і мінімізація часу затримки.

У межах кожної категорії трафік можна розподілити за пріоритетами. Високопріоритетний трафік має переваги під час обробки через його важливість для певного підприємства. Прикладом такого пріоритетного трафику може бути транзакція із замовленням. Введення пріоритетів неминуче у разі недостатності ресурсів мережі. Пріоритети можна використовувати для визначення груп, прикладних програм й окремих користувачів у групах.

Передача аудіо- і відеоінформації є чутливою до зміни затримки або джиттера. Наприклад, перевищення допустимого порогу тремтіння може призвести до достатньо відчутних спотворень зображень, необхідності дублювання відеокадрів тощо. Передача звуку також є чутливою до джиттера, тому що людині важко сприймати несподівані паузи в мовленні абонента.

Через те, що потоки аудіо- і відеоінформації проходять через різні пристрої, які обробляють трафік з урахуванням джиттера на основі різних алгоритмів, може бути швидко втрачена синхронізація між зображенням і голосом, вплив джиттера можна зменшити, використовуючи буферну пам’ять на приймальній стороні. Проте варто пам’ятати, що обсяг буфера може досягати значних розмірів, а це призводить до збільшення вартості апаратури і до небажаного ефекту – збільшення затримки за рахунок накладних витрат під час обробки інформації у великому буфері.

Огляд технологій якості обслуговування. Розрізняють велику кількість технологій, здатних забезпечити якість обслуговування в мережах. Ці технології можна умовно поділити на дві основні групи: технології, які надають неконтрольовану якість обслуговування; технології, які мають вбудовані механізми контролю за якістю обслуговування, яка надається користувачеві. Характеристики технологій якості обслуговування наведено у табл. 3.1.

Таблиця 3.1. Характеристики технологій якості обслуговування

Технологія	Переваги	Обмеження
Забезпечення достатньої пропускної здатності	Простота реалізації	За масштабованістю
Організація пріоритетних черг у маршрутизаторах	Випробована технологія, яка задовільно працює з наявними мережними протоколами	Неприйнятна для передачі аудіо- і відеоінформації в реальному часі. Вимагає додаткових ресурсів маршрутизаторів
Протокол RSVP	Може працювати в будь-яких IP-мережах	Використання не є ефективним у великих розподілених мережах
Установлення пріоритетів у віртуальних мережах	Потенційно має добре інтегруватися в мережі з комутаторами	Підтримується лише у віртуальних мережах
Технологія Frame Relay з якістю обслуговування	Використовують у комплексі з іншими технологіями, а також для передачі голосу в реальному часі	Не гарантує затримки, нестандартизована
Технологія ATM з якістю обслуговування	Випробувана стандартизована технологія	Потребує наявності магістралі ATM

Забезпечення додаткової пропускної здатності. У високошвидкісних технологіях, які з'явились останнім часом, зокрема у Gigabit Ethernet, проблеми, пов'язані з якістю обслуговування, можна розв'язати збільшенням пропускної здатності у деяких випадках, коли локальною мережею користуються невеликі робочі групи.

Використання високошвидкісних каналів зв'язку, які надають, наприклад, технології Fast/Gigabit Ethernet зі швидкостями 100/1000 Мбіт, за достатньо низького завантаження мережі дозволяє запобігти виникненню вузьких місць у мережі. Крім того, досягається й низька затримка, й незначний джиттер, причому без використання технології ATM. Головним чинником тут є прагнення відмовитися від маршрутизації й інших методів, здатних спричинити втрату пакетів і їх повторну передачу, однак, як правило, необхідно жорстко контролювати розподіл трафіку.

Методи контролю трафіку в локальних мережах є складним завданням та потребують проведення попередніх досліджень для їх ефективного впро-

вадження. У разі неконтрольованого завантаження мережі може виникнути ситуація, коли комутатори, які не справляються з навантаженням, починають втрачати пакети, які вони не можуть обробити, а кінцеві станції повторно передаватимуть ці пакети знову і знову, що ще більше збільшить об'єм трафіку. За таких умов передача аудіо- і відеоінформації в мережі є неможливою.

Під'єднання локальних мереж без централізованого керування трафіком до магістральної мережі може призвести до того, що навіть усіх резервів високошвидкісних технологій буде недостатньо для передачі пріоритетного трафіку. З огляду на це для магістральних мереж необхідно класифікувати трафік і використовувати механізм контролю якості обслуговування.

Локальні та глобальні мережі, які працюють на максимумі своєї пропускної здатності, часто зазнають труднощів через те, що немає механізмів розв'язання конфліктних ситуацій. Установка нових програм або незапланована концентрація користувачів у певній частині мережі може призвести до кризи мережі, вийти з котрої достатньо важко. Таким чином, простого збільшення пропускної здатності мережі недостатньо для її ефективної роботи.

Пріоритетні черги в маршрутизаторах. Більшість сучасних маршрутизаторів мають вбудовані функції організації пріоритетних черг, які дозволяють обслуговувати передусім певні види трафіку.

Приклад. Адміністратор може зазначити, що трафік до певної групи адрес для деяких програм повинен мати більш високий пріоритет, щоб зменшити затримку, що дуже важливо у великих розподілених мережах.

Метод пріоритетних черг найбільш часто використовують для надання часових гарантій чутливим до затримок прикладним програмам, які передають аудіо- і відеоінформацію та не потребують високої якості обслуговування. Для передачі аудіо- і відеоінформації з високою якістю слід гарантувати низьку затримку й невеликий джиттер, чого важко досягти в мережах без значних накладних витрат під час резервування буферного простору маршрутизаторів і без реалізації складних алгоритмів обробки черг.

Щодо методу пріоритетних черг немає жодного стандарту, а окремі його частини описано в різних стандартах, тому кожен виробник мережного устат-

кування реалізує у своєму обладнанні та програмному забезпеченні власні алгоритми обробки черг.

Приклад. Cisco Systems використовує алгоритм зваженої справедливої черги, а Bay Networks – чергу, в основі якої лежать класи.

Якщо маршрутизатор одержує пакети швидше, ніж він може відправити їх через певний порт, він поміщає пакети в чергу, вони надсилаються у міру надходження, тобто реалізується принцип «першим прийшов, першим пішов» – **FIFO**. Такий алгоритм достатньо ефективний, але досвід керування мережами показує, що він зовсім не оптимальний.

Випадкове раннє виявлення (Random Early Detection,) є альтернативою чергам FIFO. Цей метод дозволяє зм'якшити ефект від утрати пакетів навіть під час дуже великих навантажень. Відповідно до алгоритму RED черга як і раніше використовує принцип FIFO, але пакети відкидаються випадково, коли середня довжина черги за певний проміжок часу перевищує встановлене значення (замість того, щоб відкидати хвіст черги). Цим досягається оптимізація заповнення черги. Такий алгоритм придумано для протоколу TCP, але його можна застосовувати до трафіку будь-якого протоколу, коли мережа не гарантує доставки.

Черга з пріоритетами – це алгоритм, за якого кілька черг FIFO або RED утворюють одну чергу, трафік розподіляється між цими чергами відповідно до заданих критеріїв та відправляється в порядку суворої черговості: першим – трафік із високим пріоритетом, другим – із середнім і т. д.

Черги на основі класів (Class Based Queuing, CBQ) – це алгоритм, за якого трафік поділяють на кілька класів, кожен з яких має власну чергу і йому виокремлюється деяка частина пропускної здатності каналу.

Зважена справедлива черга (Weighted Fair Queuing, WFQ) – окремий варіант CBQ, коли класам відповідають незалежні потоки. Кожному класу призначають одну чергу FIFO та їй виокремлюється деяка частина пропускної здатності каналу, тобто відбувається перерозподіл пропускної здатності між потоками. Виділення додаткової пропускної здатності для великих потоків дозволяє зменшити затримку під час їх обробки.

Інтерфейсом до черг передачі пакетів служить протокол **резервування ресурсів (Resource Reservation Protocol, RSVP),** який дозволяє системам запитувати сервіси в мережі, наприклад гарантовану пропускну здатність, максимальний рівень втрат пакетів і передбачувану затримку.

Зазвичай цей метод справляється з перевантаженнями й допомагає запобігти появі збоїв у мережі, довів свою придатність під час «розшивки» вузьких місць мережі, однак викликає сумніви ефективність застосування цього методу у великих мережах, що обслуговують різноманітні дані. Організація черг не може усунути проблему керування безперервними потоками даних, черги організуються кожним мережним пристроєм окремо, а методу обміну інформацією між проміжними мережними пристроями, який дозволив би їм спільно впливати на швидкість її передачі, не існує. З огляду на це для черг на граничних пристроях єдиний доступний спосіб – використання буферизації пакетів, а для проміжних пристроїв – внесення помилок під час передачі з метою сповільнення потоків трафіку.

Протокол резервування ресурсів. У основу протоколу резервування ресурсів RSVP закладено три поняття: сеанс; специфікація потоку; специфікація фільтра.

Сеанс – це період обробки потоку даних, який починається з виділення ресурсів, необхідних для пропускання потоку, і закінчується після проходження потоку. Запит на резервування ресурсів від одержувача складається зі специфікації потоку й фільтра.

Специфікацію потоку визначають параметри послуг, які необхідно зарезервувати, щоб потік міг пройти через певний вузол без втрати якості.

Специфікація фільтра визначає набір пакетів, під які запитуються ресурси. Будь-які інші пакети обробляються за залишковим принципом з наданням мінімальних послуг, які мережа може забезпечити в цей час. Специфікація фільтра описує довільну підмножину пакетів одного сеансу. **Фільтр** може бути настроєний на різні параметри: конкретних відправників, конкретні протоколи, поля заголовків та інші.

Основна складність використання протоколу RSVP пов’язана з груповою розсилкою, тому що запити на виділення ресурсів поширюються у зворотному

напрямку деревом маршрутизації. Протокол RSVP використовує два основні типи повідомлень*: RESV* й *PATH.*

Перше (*RESV*) повідомлення генерується одержувачами й поширюється вгору деревом маршрутизації, передається кожному маршрутизатору на шляху потоку. Якщо маршрутизатор не може забезпечити необхідну смугу пропускання, то він відкидає цей запит, якщо може – це повідомлення передається наступному маршрутизатору. Таке повідомлення *RESV* переводить маршрутизатор у стан резервування ресурсів для сеансу. Об'єднані повідомлення *RESV* досягають відправника, і на підставі отриманої інформації про зарезервовані ресурси відправник задає необхідні параметри потоку першого маршрутизатора.

Повідомлення *PATH* використовується для поширення інформації про зворотний маршрут. Використовуваний протоколом RSVP алгоритм маршрутизації не може визначити зворотний маршрут, а повідомлення *RESV* має передаватися саме зворотним маршрутом. Інформація про зворотний маршрут отримується у такий спосіб: будь-який пристрій, що бажає стати відправником, надсилає членам своєї групи повідомлення *PATH*, одержавши яке маршрутизатори і члени групи переходять у стан *PATH*. У цьому стані пакети для цього відправника мають пересилатися на маршрутизатор, від якого їх було отримано. Кожен маршрутизатор, який одержав повідомлення *PATH*, запам'ятовує ідентифікатор потоку й канал зв'язку, яким надійшло це повідомлення. Якщо потенційний адресат, який прийняв команду *PATH*, хоче одержати зазначені дані, то він надсилає команду *RESV*, яка йде шляхом *PATH*, але у зворотному напрямку.

Приклад. Розглянемо спрощений опис такого алгоритму. Припустімо, відправник, знаючи всіх одержувачів, хоче показати їм відеокліп. Оскільки адреси відомі, він надсилає їм повідомлення *PATH,* яке містить інформацію про те, що відправник збирається показати відеокліп членам своєї групи. Шляхом проходження до адресатів повідомлення виставляє на маршрутизаторах необхідні для передачі потоку параметри. Якщо який-небудь маршрутизатор не може забезпечити цих параметрів, то він відкидає повідомлення, і це означає, що одержувач на такому маршруті не зможе подивитися відеокліп. Коли повідомлення *PATH* отримали всі одержувачі, вони аналізують прийняту із цим повідомленням інформацію й відповідають повідомленням *RESV,* яке проходить маршрутом

повідомлення *PATH*, але у зворотному напрямку. Одержувачі закладають у повідомлення *RESV* інформацію про те, чи хочуть вони переглянути кліп. Вони можуть попросити показати інший кліп, зменшити якість відео, тому що мають погані канали зв'язку тощо. Після того, як відправник одержав усі повідомлення *RESV*, він починає сеанс, ураховуючи побажання кожного одержувача.

Повідомлення *RESV/PATH* можна використовувати для визначення вузла або каналу зв'язку, який вийшов з ладу. Обмін цими повідомленнями підтверджує, що сеанс ще не завершено, тобто виділені ресурси мають підтримуватися.

Приклад. Розглянемо інший приклад роботи протоколу RSVP. Щоб зарезервувати смугу пропускання для якого-небудь трафіку, робоча станція надсилає найближчому маршрутизатору запит протоколу RSVP, який визначає необхідний канал із пропускною здатністю (наприклад, 1 Мбіт/с) до певного одержувача. Цей запит отримують усі маршрутизатори на шляху. Якщо маршрутизатор може виконати вимоги, зазначені в запиті, він передає запит наступному маршрутизатору й т. д. Запит вважається виконаним, якщо всі маршрутизатори підтвердили виконання вимог. Якщо один з маршрутизаторів не може виконати вимоги, то він надасть кінцевій станції повідомлення про те, що запит відхилено.

Основний недолік протоколу RSVP полягає у тому, що наявні протоколи маршрутизації не беруть до уваги проблему якості обслуговування, а запити RSVP виконуються тільки після того, як було обрано маршрут передачі. З деякої кількості альтернативних маршрутів протокол маршрутизації може обрати маршрут, не оптимальний з погляду протоколу RSVP, що призводить до того, що цей маршрут найчастіше не може відповідати запиту RSVP через, наприклад, брак резервів пропускної здатності. Оскільки всі маршрутизатори на шляху передачі даних мають урахувати запити RSVP, подібні розглянутій ситуації можуть призводити до відхилень запитів, хоча якби було обрано інший шлях, запити було б виконано.

Нездатність протоколів маршрутизації враховувати питання якості обслуговування під час вибору маршруту зумовлюють іншу проблему, пов'язану з протоколом RSVP, – масштабованість. Кожен маршрутизатор має приймати й обробляти інформацію про стан усіх контрольованих протоколом RSVP потоків даних, що проходять через нього, а це може призвести до перевантаження маршрутизатора, який обслуговує занадто багато потоків. Проблема тільки частково розв'язується вмінням RSVP поєднувати марш-

рути для групової доставки даних. Труднощі також виникають у процесі обробки даних, які перенапрямляються іншими маршрутами, наприклад через виникнення перевантажень. Нові маршрути мають бути прокладені відповідно до вимог RSVP за допомогою спеціальних повідомлень від інших маршрутизаторів у потоці.

Незважаючи на ці недоліки й той факт, що протокол RSVP перебуває у стадії розгляду в групі IETF і не є ефективним у великих розподілених мережах, усі провідні виробники мережного устаткування (Cisco Systems, 3Com, Bay Networks та інші) використовують його у своїх маршрутизаторах.

3.4.3 Синхронізація потоків даних

Під **синхронізацією потоків даних** розуміють підтримку часових співвідношень між ними. Розрізняють два типи синхронізації: найпростіший тип синхронізації відбувається між **дискретними** й **безперервними** потоками даних; більш складний тип синхронізації - між безперервними потоками даних.

Синхронізація проходить на рівні елементів даних, з яких складається потік, тобто можна синхронізувати два потоки тільки відносно елементів даних. Вибір елементів даних залежить від рівня абстрагування представлення потоку даних.

Приклад. Розглянемо аудіопоток (одноканальний) з якістю компакт-диску. За максимального ступеня деталізації потік даних подають у вигляді послідовності 16-бітних вибірок. За частоти вибірки 44100 Гц синхронізація з іншим аудіопотоком може теоретично відбуватися кожні 23 мс. Щоб одержати високоякісний стереоефект, синхронізація на цьому рівні вкрай необхідна.

Механізми синхронізації. Варто розглянути два моменти: базові механізми синхронізації двох потоків даних і розподіл цих механізмів у мережному середовищі.

Механізми синхронізації можна розглядати з позицій різних рівнів абстрагування: згідно з найнижчим синхронізація повністю визначається роботою з елементами даних простих потоків. Цей принцип ілюструє рис. 3.8, на якому

показано процес, який здійснює операції читання й запису в декількох простих потоках даних, гарантуючи при цьому забезпечення обмежень синхронізації.

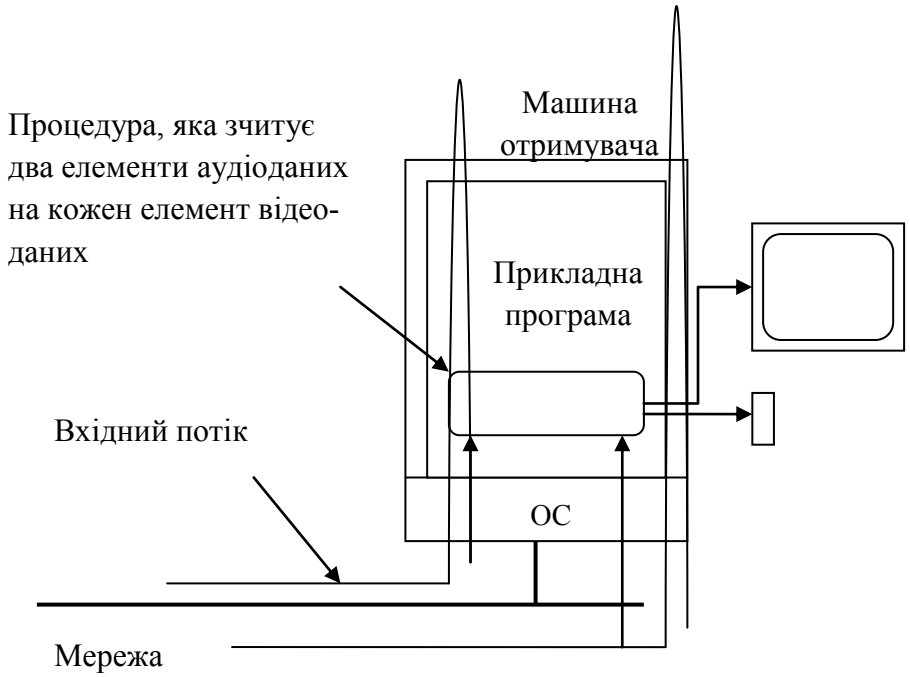


Рис. 3.8. Принцип явної синхронізації на рівні елементів даних

Приклад. Розглянемо фільм, представлений двома вхідними потоками даних. Відеопотік містить нестиснені зображення низької якості з роздільною здатністю 320x240 пікселів, кожний з яких під час кодування потребує 1 байт, що в сумі утворює елементи відеоданих розміром 76 800 байт. Припустімо, зображення демонструються із частотою 30 Гц, або по 33 мс на зображення. Вважаємо, що аудіопотік містить аудіовибірки, згруповані в елементи довжиною 11760 байт, кожний з яких відповідає звуку тривалістю 33 мс. Якщо процес введення забезпечує швидкість 2,5 Мбайт/с, то можна синхронізувати потоки, просто чергуючи читання зображень і блоків аудіовибірок кожні 33 мс.

Однак прикладне програмне забезпечення може повністю відповідати за синхронізацію лише в тому разі, якщо воно має доступ до механізмів низького рівня. Отже, бажано прикладному програмному забезпеченню надати інтерфейс, який спростив би керування потоками і пристроями.

Приклад. Стосовно передачі фільму це буде означати, що відеодисплей матиме інтерфейс керування, за допомогою якого зможе керувати частотою відтворення зображень. Окрім того, інтерфейс надасть механізм реєстрації оброблювача дій користувача, що викликатиметься щоразу після прийому чергових k зображень, аналогічний інтерфейс буде наданий і пристрою відтворення звуку. Використовуючи ці інтерфейси керування, розробник прикладного програмного забезпечення зможе написати просту програму монітори-

нгу, яка порівнює два заголовки відповідних потоків даних, перевіряє, чи достатня синхронізація відео- та аудіопотоків і, якщо виникає потреба, регулює частоту передачі відео- або аудіоелементів. Цей приклад (рис. 3.9) типовий для багатьох систем мультимедіа, які використовують як прикладне програмне забезпечення проміжного рівня.

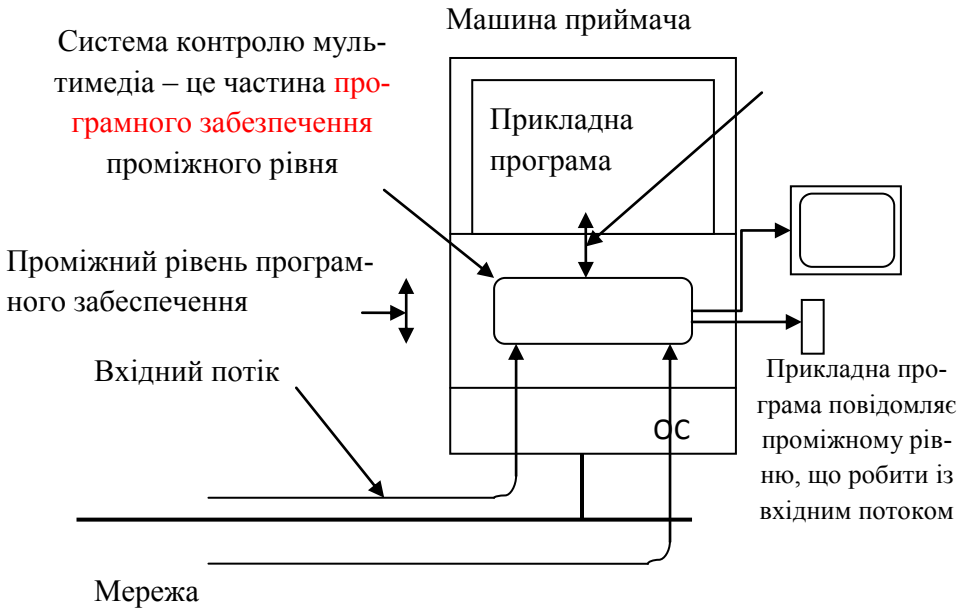


Рис. 3.9. Принцип синхронізації, підтримуваний високорівневими інтерфейсами

Одержувач комплексного потоку даних повинен мати повну локальну специфікацію синхронізації, але зазвичай ця інформація надається в неявному вигляді, за рахунок мультиплексування декількох потоків даних в один, який містить усі елементи даних, у тому числі й призначені для синхронізації.

3.5 Протоколи проміжного рівня

3.5.1 Протокол SOAP

Протокол SOAP (Simple Object Access Protocol) є протоколом проміжного рівня за моделлю OSI, тобто він визначає формат повідомлень, що пересилаються за допомогою деякого транспортного протоколу, яким зазвичай є HTTP, HTTPS, TCP, іноді SMTP.

Формат повідомлень SOAP грунтується на XML, SOAP-повідомлення складається з таких частин:

– **конверт** *(envelope)* – містить повідомлення цілком;

– **заголовок** *(header)* – містить значення деяких додаткових атрибутів повідомлення, використовуваних під час його обробки або переадресації. Заголовку може не бути або його можуть використовувати зазвичай для передачі інформації про координацію декількох повідомлень, ідентифікацію сеансів і передачі різних сертифікатів для захисту інформації;

– **тіло** *(body)* – основний вміст повідомлення, який має належати одному з типів повідомлень, якими можна обмінюватися з певною службою відповідно до опису її інтерфейсу. Тіло має бути в будь-якому повідомленні.

Простий приклад SOAP-повідомлення наведено нижче.

```
<SOAP-ENV: Envelope
 xmlns: SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV: Header>
  <t: Transaction
   xmlns: t="http: //company.com/soap-headers/attrs "
   SOAP-ENV:mustUnderstand="1">
  </t: Transaction>
 </SOAP-ENV: Header>
 <SOAP-ENV: Body>
  <m: GetLastTradePrice xmlns: m="http://company.com/web-services/ trading">
   <symbol> DEF</symbol>
  </m: GetLastTradePrice>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Крім формату повідомлень, протокол SOAP визначає процес їх обробки різними посередниками й одержувачами.

3.5.2 Сімейство протоколів XMPP

Сімейство **протоколів XMPP** (Extensible Messaging and Presence Protocol) прийнято як стандарт RFC (Request for Comments), стандартний порт для XMPP – 5222, але також використовують порт 80 та/або 443, якщо виникають проблеми у процесі роботи з брандмауером (firewall).

Переваги сімейства протоколів XMPP, такі:

– *Децентралізація.* Архітектура мережі XMPP аналогічна електронній пошті, тобто будь-хто може запустити власний XMPP-сервер, не використовуючи центрального.

– *Відкритий стандарт*. Інженерна рада Internet (IETF - Internet Engineering Task Force) формалізувала XMPP як стандарт миттєвого обміну повідомленнями й технологію присутності під назвою XMPP; специфікації XMPP було опубліковано як стандарти RFC 3920 і RFC 3921. Розробка цих специфікацій не прив'язана до якогось розробника. Існує багато реалізацій серверів і клієнтів, а також бібліотек з відкритим вихідним кодом.

– *Історія.* Технологію XMPP впроваджено 1998 року за підтримки таких великих компаній, як Sun Microsystems і Google, створено значну кількість доповнень до стандартів XMPP для клієнтів, серверів, компонентів і бібліотек кодів.

– *Безпека*. XMPP-сервери можуть бути ізольованими від публічних мереж XMPP (наприклад у внутрішній мережі компанії) та добре захищеними (завдяки використанню механізмів SASL (Simple Authentication and Security Layer) і TLS (Transport Layer Security)), вбудованими в ядро XMPP специфікаціями. Для можливості використання шифрування каналу XMPP Standards Foundation також застосовують додаткову службу *certification authority* в *xmpp.net*, забезпечуючи цифрові сертифікати для адміністраторів XMPP серверів у разі використання *StartCom Certification Authority*. Багато реалізацій серверів застосовують *SSL* під час обміну між клієнтом і сервером, чимало клієнтів мають можливість шифрування за допомогою *PGP/GPG* усередині протоколу.

– *Гнучкість*. Функціональність, яка налаштовується, може бути надбудованою поверх XMPP для підтримки взаємодії різних мереж, стандартні розширення підтримують *XMPP Software Foundation*.

Недоліки сімейства протоколів XMPP, такі:

– *Надлишковість інформації, що передається*. Більше 70 % міжмережного трафіку XMPP стосується повідомлень зі службовою інформацією, близько 60 % яких є зайвими. На сьогодні XMPP створює надлишковий трафік під час доставки «статус-повідомлень» (повідомлень про присутність)

декільком користувачам. Для розв'язання цієї проблеми розробляють нові протоколи, розширяють XEP-0138 (компресія переданих даних протоколу алгоритмами *lzw* і *zlib*), а також використовують компресію в межах шифрування з'єднання TLS RFC 3749;

– *Масштабованість*. XMPP має також проблему надмірності, але стосовно чат-кімнат і можливостей публікації інформації. Розв'язання цієї проблеми передбачається у застосуванні XEP-розширень.

– *Неефективність передачі бінарних даних*. Оскільки XMPP є одним XML-документом, неможливо передавати немодифіковану двійкову інформацію, тому для передачі файлів застосовують додаткові протоколи, наприклад HTTP, а для передачі файлів й іншої бінарної інформації безпосередньо в XMPP-потоці використовують кодування *base64*.

Адресація. Кожен користувач у мережі Jabber має унікальний ідентифікатор – JID (*Jabber ID*). Адреса JID, подібна адресі електронної пошти, містить ім'я користувача й доменне ім'я сервера, на якому зареєстровано користувача, розділені знаком @.

Приклад. Користувач *user*, зареєстрований на сервері *example.com*, матимеме адресу: *user@example.com*.

Користувач може мати одночасно кілька підз'єзднань, для розрізнення яких використовують додаткові значення JID, які називають ресурсом та додають через знак «/» у кінці адреси.

Приклад. Нехай повна адреса користувача буде *user@example.com/work*, тоді повідомлення, надіслані на адресу *user@example.com*, дійдуть на зазначену адресу незалежно від імені ресурсу, але повідомлення для *user@example.com/work* дійдуть на зазначену адресу тільки за умови відповідності під'єднаному ресурсу.

Адреси JID можуть також використовуватися без явного визначення імені користувача (з визначенням імені ресурсу або без такого) для системних повідомлень і для контролю спеціальних можливостей на сервері.

З'єднання з іншими протоколами. Корисною особливістю XMPP-систем є транспорти, або **шлюзи**, які дозволяють користувачам отримувати доступ до мережі, побудованої на основі інших протоколів, зокрема протоколів миттєвого обміну повідомленнями або SMS і електронної пошти.

На відміну від мультипротокольних клієнтів, XMPP надає доступ на рівні сервера за допомогою комунікації через спеціальні сервіси-шлюзи, що функціонують на віддаленому комп'ютері.

Будь-який користувач може «зареєструватися» на одному із цих шлюзів, надавши інформацію, необхідну для входу в мережу, і спілкуватися з користувачами мережі так, ніби вони є користувачами мережі Jabber. Це означає, що будь-яке програмне забезпечення – клієнт, який підтримує XMPP, може бути використаний для доступу до будь-якої мережі, для якої наявні шлюзи, не вводячи додаткового коду та не потребуючи прямого доступу до Internet від клієнта. Реалізація шлюзів залежить від конкретного XMPP-сервера та є нестабільною через закритість комерційних IM-сервісів.

3.5.3 Протокол UMSP

Протокол **UMSP** (Unified Memory Space Protocol) відповідає проміжному рівню моделі OSI, призначений для реалізації в широкому класі систем: від простих пристроїв на основі спеціалізованих процесорів до універсальних комп'ютерів та кластерів. Для обміну даними протокол використовує сервіс транспортного рівня з гарантованою доставкою, а для передачі даних не потрібні підтвердження, тому використання протоколу не забезпечує гарантованої доставки.

Головна функція протоколу UMSP – створення мережного середовища для організації 128-розрядного адресного простору пам'яті, розподіленого між вузлами Internet, тобто протокол регламентує механізм установлення з'єднання і формат переданих інструкцій, але не здійснює безпосереднього керування локальною пам'яттю на вузлі.

На відміну від традиційних мережних протоколів, в яких під час додавання користувача на різних вузлах здійснюється взаємодія за допомогою обміну мережними примітивами або роботи з потоками даних, в даному протоколі взаємодія здійснюється за допомогою безпосереднього читання/ запису даних або передачі керування коду у віртуальній пам'яті віддаленого вузла. Користувач, якого додали, може нічого не знати про наявність протоколу в мережі, а просто користуватися інструкціями з 128-розрядними адресами.

Протокол UMSP, передусім, використовують у системах на основі віртуальних машин (VM), які обробляють псевдокод, однак його можна застосовувати в системах, що безпосередньо виконують код процесора, наприклад, у кластерах або в універсальних операційних системах для організації розподіленого віртуального адресного простору. Крім того, мінімальний профіль протоколу можна використати у простих пристроях, що не мають операційної системи.

Протокол надає гнучкі ресурси для створення параметрів з'єднань і дозволяє будувати системи з високим рівнем захисту без негативного впливу на функціональні можливості прикладного програмного забезпечення, може істотно спростити процес розробки розподілених систем, а також поєднує не тільки інформаційні, але й обчислювальні ресурси великої кількості різнотипних комп'ютерів без значних витрат на стандартизацію і розробку програмного забезпечення.

3.6 Висновки

1. Наявність потужних і гнучких механізмів взаємодії між процесами є важливою для будь-якої розподіленої системи. У традиційних мережних прикладних програмах зв'язок часто грунтується на низькорівневих примітивах передачі повідомлень, які надаються транспортним рівнем. Важливою особливістю систем проміжного рівня є надаваний ними високий ступінь абстракції, завдяки якому опис взаємодії між процесами на проміжному рівні значно спрощується. Такий підхід є більш ефективним, ніж якщо обмежитися лише інтерфейсами транспортного рівня.

2. Щоб спростити роботу зі значною кількістю рівнів і понять, використовуваних під час передачі даних, Міжнародна організація стандартизації ISO розробила еталонну модель, яка чітко визначає рівні абстракції, дає їм стандартні імена і зазначає, який рівень за що відповідає. Цю модель названо Еталонною моделлю взаємодії відкритих систем OSI та розроблено для того, щоб надати відкритим системам можливість взаємодіяти.

3. Однією з найбільш широко використовуваних абстракцій є віддалений виклик процедур RPC. Сутність RPC полягає у реалізації будь-якої

служби за допомогою виклику процедури, тіло якої виконується на сервері. Клієнт надає тільки сигнатуру процедури, тобто її ім’я та параметри.

4. Механізм орієнтований на забезпечення прозорості доступу, однак він порівняно слабко підтримує передачу посилань. З цього погляду віддалені об’єкти більш прозорі. Звертання до віддалених методів RMI нагадує RPC, але враховує специфіку віддалених об’єктів. Основна відмінність між методами полягає в тому, що RMI дозволяє використати як параметри посилання на об’єкти системи.

5. Процедури RPC і RMI надають механізми синхронного зв’язку, в разі якого клієнт блокується до одержання відповіді від сервера.

6. У моделях передачі повідомлень неважливо, чи є зв’язок збереженим або синхронним. Суть збереженого зв’язку полягає в тому, що повідомлення, яке надсилається, зберігається в комунікаційній системі доти, доки не буде доставлено за призначенням. У разі нерезидентного зв’язку механізмів зберігання не передбачено, а тому одержувач має бути готовий прийняти повідомлення у будь-який момент.

7. У разі асинхронного зв’язку відправник може продовжувати роботу відразу після того, як повідомлення поставлено в чергу на відправлення, можливо, ще до того, як його буде відправлено. У разі синхронного зв’язку відправник блокується до моменту одержання повідомлення.

8. Моделі обміну повідомленнями проміжного рівня зазвичай надають збережений асинхронний зв’язок і використовуються там, де застосування механізмів RPC і RMI не виправдано, передусім у разі інтеграції наборів баз даних (сильно розподілених) у великих інформаційних системах. Інші області їх застосування є електронна пошта й робочі потоки.

9. Абсолютно інший зв’язок використовують для потоків даних, проблема яких полягає в тому, що будь-які два послідовні повідомлення взаємозалежні за часом. У безперервних потоках даних затримка доставки для кожного повідомлення є максимальною, а під час синхронізації безперервних потоків даних повідомлення має мати певну мінімальну затримку доставки. Типовими прикладами безперервних потоків даних є аудіо- і відеопотоки. Ускладнюючим фактором є роль джиттера (максимальне й мінімальне значення його параметрів). Навіть якщо середня продуктив-

ність є досяжною, то серйозні коливання часу доставки можуть призвести до неприйнятної продуктивності.

10. До сучасних протоколів проміжного рівня належать протокол SOAP, родина протоколів XMPP, протокол UMSP та інші. Протокол SOAP визначає формат повідомлень, які пересилаються за допомогою деякого транспортного протоколу, наприклад HTTP, TCP, іноді SMTP. Родину протоколів XMPP було розроблено як родину стандартів миттєвого обміну повідомленнями й технологію присутності, а протокол UMSP розроблено для організації обміну повідомлень у 128-розрядному адресному просторі пам'яті.

3.7 Запитання для самоконтролю

1. Призначення й рівні моделі OSI.
2. Низькорівневі протоколи, їх особливості.
3. Транспортні протоколи, TCP/IP.
4. Назвіть протоколи верхнього рівня, їхнє призначення.
5. Метод RPC, його застосування.
6. Звертання до віддалених методів RMI.
7. Види звертання до віддалених об'єктів.
8. Відмінності зв'язку за допомогою повідомлень і на основі потоків даних.
9. Основні механізми синхронізації потоків даних.
10. Синхронізація потоків даних.
11. Що таке сеанс?
12. Наведіть приклади пріоритетних черг у маршрутизаторах.
13. Назвіть протоколи проміжного рівня.
14. Назвіть основні частини протоколу SOAP. Для чого він використовується?
15. Розкрийте позитивні та негативні сторони сімейства протоколів XMPP.
16. Коротко охарактеризуйте протокол UMSP.

4. ПРОЦЕСИ

4.1 Поняття процесу. Визначення і структура

Комп'ютерна програма – це пасивна сукупність інструкцій, у той час як **процес** – це безпосереднє виконання цих інструкцій.

Для виконання програм операційна система створює декілька *віртуальних процесорів* окремо для кожної програми. Щоб відстежувати ці віртуальні процесори, операційна система підтримує *таблицю процесів* (process table), яка містить записи для збереження значень регістрів процесора, карт пам'яті, відкритих файлів, облікових записів користувачів, привілеїв тощо.

Процес (process) часто визначають як *виконувану програму*, тобто таку, яка в певний час виконується на одному з віртуальних процесорів операційної системи. Слід зазначити, що операційна система гарантує те, щоб ці незалежні процеси спеціально або ненавмисно не порушили правильної роботи інших процесів, тобто певні процеси спільно і прозоро використовують один і той самий процесор та інші апаратні ресурси. Зазвичай для того, щоб відокремлювати процеси один від одного, операційній системі потрібна апаратна підтримка.

Прозорість паралельної роботи коштує достатньо дорого з погляду ресурсів обчислювальної системи. Так, кожного разу під час створення процесу операційна система має створювати *абсолютно незалежний адресний простір*. Виділення пам'яті вимагає ініціалізації сегмента пам'яті, наприклад, за рахунок обнулення сегмента даних, копіювання відповідної програми в сегмент коду і розміщення її у стек тимчасових даних. Перемикання процесора між двома процесами – також досить ресурсоємна операція, під час якої, крім збереження контексту процесора (у який входять значення регістрів, лічильник програми, вказівник на стек тощо), операційна система має також змінити регістри блока керування пам'яттю (Memory Management Unit, MMU) й оголосити некоректним вміст кешу трансляції адрес, наприклад *асоціативного буфера сторінок* (Translation Lookaside Buffer, TLB). Якщо операційна система підтримує більше процесів, ніж може одночасно міститися в оперативній пам'яті, то перед дійсним перемиканням з одного процесу

на другий може виникнути потреба *підкачки* (swapping) процесів між оперативною пам’яттю і диском.

Процес складається з трьох сегментів: *сегмента коду, сегмента ресурсів і сегмента виконання* (рис. 4.1).

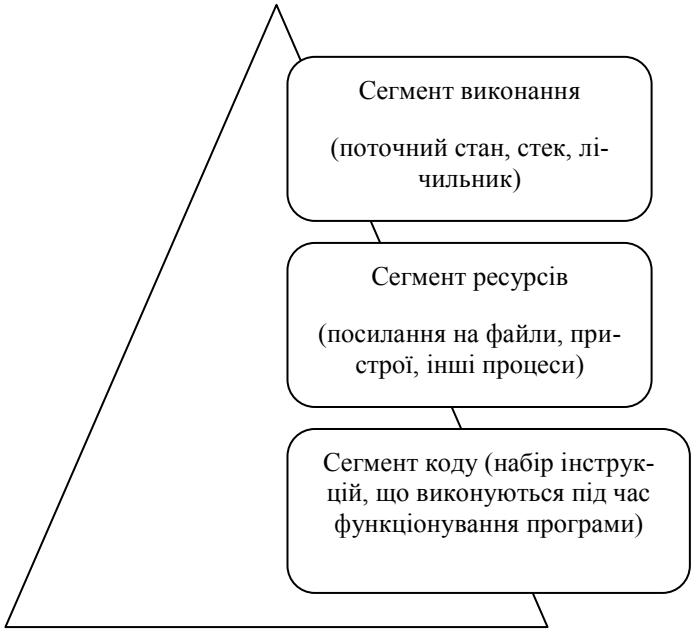


Рис. 4.1. Структура процесу

Сегмент коду – це частина, що містить набір інструкцій, які виконуються у процесі виконання програми. **Сегмент ресурсів** містить посилання на зовнішні ресурси, необхідні процесу, зокрема на файли, принтери, пристрої, інші процеси. **Сегмент виконання** використовується для зберігання поточного стану процесу, включаючи закриті дані, стек і лічильник програми.

4.2 Потоки виконання. Визначення і структура

Незважаючи на те, що процеси є будівельними блоками розподілених систем, практика показує, що надмірний поділ на процеси негативно впливає на продуктивність системи. Натомість поділ у вигляді декількох **потоків виконання** (threads) в одному процесі суттєво спрощує побудову розподілених програмних систем і дозволяє досягти кращої продуктивності. Розглянемо роль потоків виконання в розподілених системах.

Будь-який потік складається з двох компонентів: *об'єкта ядра*, через який операційна система керує потоком і в якому зберігається статистична інформація про потік (додаткові потоки створюються також ядром); *стека потоку*, який містить параметри всіх функцій і локальні змінні, необхідні потоку для виконання коду.

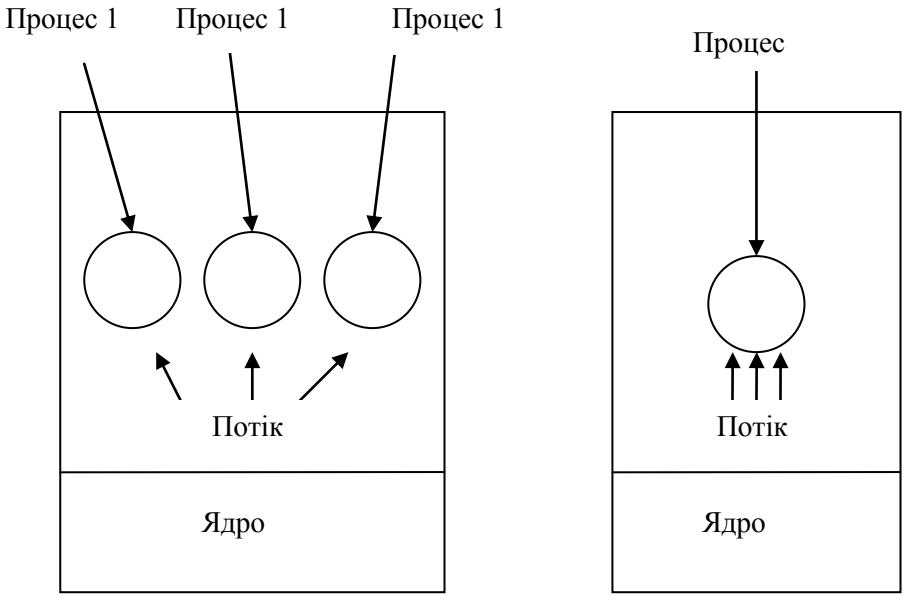


Рис. 4.2. Структура потоку виконання

Основна відмінність процесів від потоків полягає в тому, що *процеси, ізольовані один від одного*, функціонують у різних адресних просторах, а потоки можуть використовувати *один і той самий простір* (усередині процесу), виконуючи дії та не заважаючи один одному.

Потік виконання є подібним процесу, зокрема тим, що його можна розглядати як програму, яка виконується на віртуальному процесорі, але, на відміну від процесу, не можна досягти високої міри прозорості паралельного виконання потоків, оскільки це призводить до спадання продуктивності. Система потоків виконання зазвичай забезпечує лише той мінімум інформації, який дозволяє спільно використовувати процесор для різних потоків виконання, зокрема *контекст потоку виконання* (thread context) іноді містить контекст процесора і деяку іншу інформацію, необхідну для керування потоком виконання.

Приклад. Система потоків може відстежувати факт блокування потоку змінною алгоритму, наприклад *«Mutual execution»,* і зумовлену цим неможливість перемикання на виконання такого потоку.

Інформація, що не є абсолютно необхідною для керування численними потоками, зазвичай ігнорується. З цієї причини завдання захисту даних від несанкціонованого доступу у потоках виконання одного процесу покладено на розробників прикладного програмного забезпечення.

У такому підході є два аспекти: по-перше, високої продуктивності багатопотокових програмних засобів можна досягти з набагато меншими зусиллями, ніж у разі їх однопотокового аналога, тобто багатопотокові системи зазвичай дають виграш у продуктивності; по-друге, оскільки потоки виконання одного процесу не захищаються автоматично один від одного, розробка багатопотокових програмних засобів потребує додаткових зусиль.

4.3 Стан процесів та потоків виконання

У багатозадачній (багатопроцесорній) системі процес може перебувати в одному з трьох основних станів:

– **виконання** – активний стан процесу, в якому процес має всі необхідні ресурси і безпосередньо виконується процесором;

– **очікування** – пасивний стан процесу, в якому процес заблокований, не може виконуватися через внутрішні причини, тому він чекає здійснення певної події, наприклад завершення операції введення-виведення, отримання повідомлення від другого процесу, звільнення якого-небудь необхідного йому ресурсу;

– **готовність** – також пасивний стан процесу, але в цьому разі процес заблокований у зв’язку із зовнішніми відносно нього обставинами: процес має всі необхідні для нього ресурси, він готовий виконуватися, однак процесор здійснює виконання другого процесу.

Протягом життєвого циклу кожен процес переходить з одного стану в другий відповідно до алгоритму планування процесів, реалізованого в операційній системі. Типовий граф станів процесу показано на рис. 4.3.

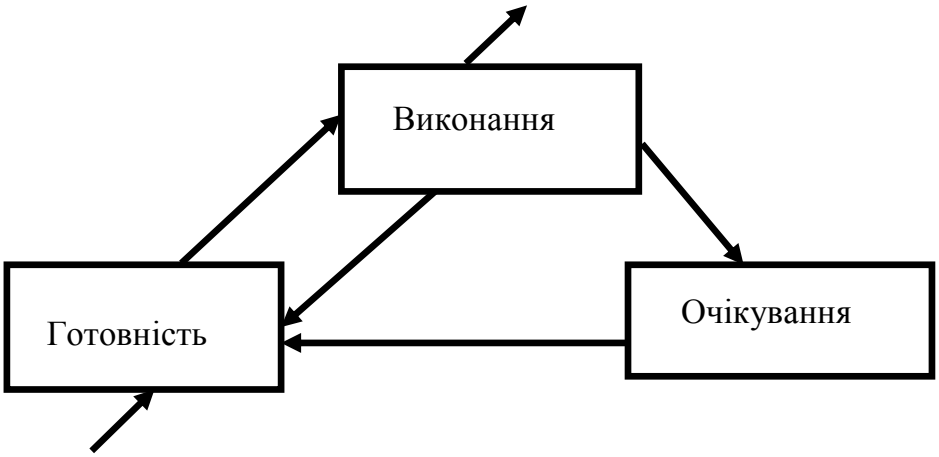


Рис. 4.3. Типовий граф станів процесів

У стані виконання в однопроцесорній системі може перебувати лише один процес, а в кожному зі станів очікування і готовності – кілька процесів, які утворюють відповідні *черги процесів у стані готовності й очікування*.

Життєвий цикл процесу починається зі стану готовності, коли процес готовий до виконання і чекає своєї черги. У разі активізації процес переходить у стан виконання й перебуває в ньому доти, поки він сам звільнить процесор, перейшовши у стан очікування якої-небудь події, або його буде насильно «витиснено» із процесора, наприклад унаслідок вичерпання відведеного цьому процесу кванта процесорного часу, в цьому разі процес повертається у стан готовності. У цей же стан процес переходить зі стану очікування після того, як очікувана подія відбудеться.

4.4 Реалізація потоків виконання

Розрізняють два підходи реалізації потоків виконання: *статичний* і *динамічний.* За **статичним підходом** кількість потоків виконання визначають уже на стадії написання програми або на стадії компіляції; кожному потоку виконання призначається *фіксований стек*. Цей підхід простий, але негнучкий.

Більш загальним є **динамічний підхід**, який дозволяє створювати і видаляти потоки виконання *оперативно під час виконання*. Системний виклик для створення потоку виконання зазвичай міститься в потоці головної програми у

вигляді вказівника на процедуру із зазначенням розміру стека, а також інших параметрів, наприклад диспетчерського пріоритету. Виклик зазвичай повертає *ідентифікатор потоку виконання*, який можна використовувати в подальших викликах, пов'язаних із цим потоком. У цій моделі процес починається з одного потоку, але може створювати їх більше у разі потреби.

Завершуватися потоки можуть одним з двох способів: *за власною ініціативою*, коли завершується робота потоку, та *ззовні*. У багатьох випадках, наприклад за конвеєрною моделлю, потоки виконання створюються відразу після початку процесу і ніколи не знищуються.

4.4.1 Потоки виконання в нерозподілених системах

Найбільш важлива перевага використання потоків виконання в нерозподілених системах полягає у тому, що процеси з одним потоком виконання *цілком блокуються* під час будь-якого блокуючого системного виклику.

Приклад. Як ілюстрацію розглянемо певне програмне забезпечення, наприклад електронні таблиці, важливою властивістю яких є підтримка функціональних залежностей між різними елементами таблиці або різними таблицями. Таким чином, кожного разу в процесі модифікації комірки всі комірки, пов'язані з нею, автоматично оновлюються. Уявимо, що користувач хоче постійно змінювати значення комірки в інтерактивному режимі. Після того як користувач змінить значення однієї з комірок, внесена ним зміна спричинить значний обсяг обчислень. Якщо працювати лише з одним потоком виконання, то під час очікування введення даних ці обчислення виявляться неможливими. Так само складно ввести дані під час виконання процесу обчислень. Найпростішим рішенням є створення як мінімум двох потоків виконання: одного - для підтримки роботи з користувачем, другого - для оновлення таблиць.

Багатопотокову структуру часто використовують, будуючи великі програмні комплекси, які часто розробляють у вигляді наборів спільно працюючих програм, кожна з яких виконується окремим процесом. Цей підхід типовий для середовища UNIX. Кооперація між програмами реалізується у вигляді міжпроцесної взаємодії (через *механізми Interproccessing Cooperation* – IPC). Для UNIX-систем ці механізми зазвичай застосовують *канали (поіменовані)*, *черги повідомлень* і *спільно використовувані сегменти пам'яті.* Основною

проблемною властивістю механізмів IPC є необхідність інтенсивного перемикання контекстів, яка продемонстрована трьома точками на рис. 4.4.

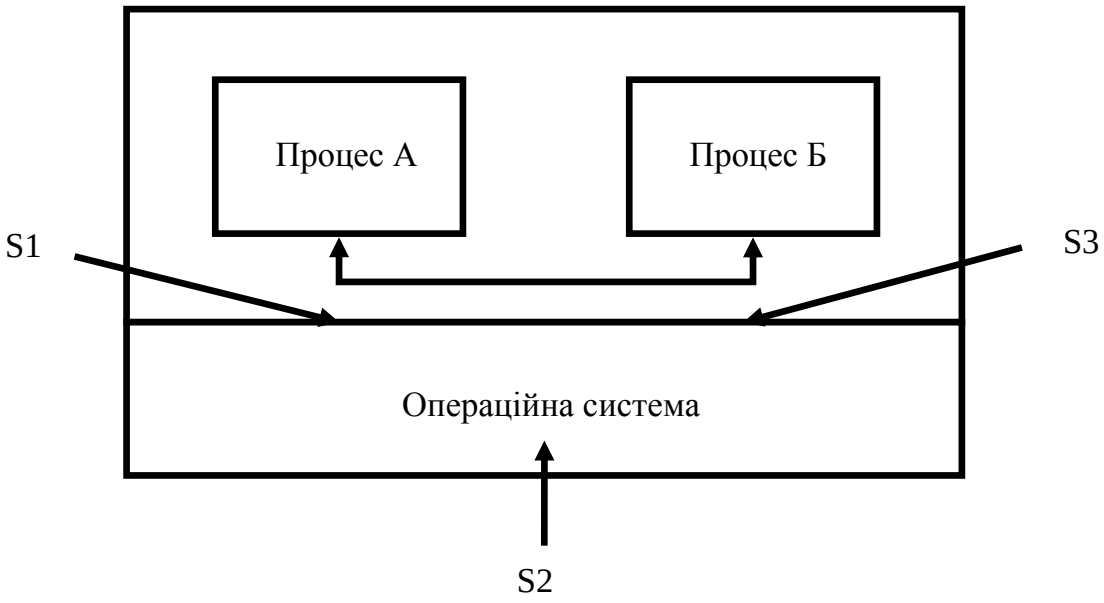


Рис. 4.4. Перемикання контекстів у результаті виклику IPC

Оскільки механізм IPC має втручатися в ядро, процес зазвичай вимушений спочатку перемкнутися з призначеного для користувача режиму в режим ядра (точка *S1*), що потребує зміни карти пам'яті у блоці MMU, а також очищення буфера TLB. У ядрі відбувається перемикання контексту процесу (точка *S2*), після чого другий процес може бути активізований черговим перемиканням з режиму ядра у призначений для користувача режим (точка *S3*). Останнє перемикання теж потребує зміни карти пам'яті у блоці MMU й очищення буфера TLB.

Потоки виконання застосовують розробники прикладних систем через те, що багато складних програмних комплексів дуже легко розробляти, структурувавши їх у вигляді набору взаємозв'язаних потоків виконання.

Приклад. Розглянемо прикладне програмне забезпечення, що вимагає виконання декількох незалежних завдань. Так, для текстового редактора в окремі потоки можна виокремити обробку введення даних користувача, перевірку орфографії та граматики, оформлення зовнішнього вигляду документа, створення індексу тощо.

4.4.2 Потоки виконання в розподілених системах

Важливою властивістю потоків виконання в розподілених системах є зручна реалізація *блокуючих системних викликів*, які відбуваються *без блокування всього процесу* на час виконання потоку. Ця властивість потоків виконання особливо важлива в розподілених системах, оскільки вона суттєво спрощує представлення взаємодії як одночасну підтримку великої кількості логічних з'єднань.

4.4.3 Багатопотокові клієнти

З метою підвищення прозорості розподілу розподілені системи, які працюють у глобальних мережах, потребують маскування значних затримок повідомлень, які курсують між процесами. *Цикл затримки* у глобальних мережах може досягати сотень мілісекунд або і секунд. Традиційний спосіб приховати затримки зв'язку – ініціювавши взаємодію, відразу розпочати виконання другого завдання.

Приклад. Типовим варіантом використання цієї методики є web-браузери. Здебільшого web-документ у файлі формату HTML містить окрім тексту набір зображень, значків тощо. Для отримання елементів web-документа браузер відкриває з'єднання TCP/IP, читає дані, що надходять, і перетворює їх у компоненти візуального представлення. Встановлення з'єднання, як і читання даних, є блокуючою операцією. У разі роботи з повільними комунікаціями відчуваються незручності, пов'язані з тим, що час, потрібний для завершення кожної операції, може бути достатньо тривалим.

Web-браузер зазвичай спочатку отримує сторінку HTML-коду, а потім показує її. Для того, щоб приховати затримки зв'язку деякі браузери починають показувати дані у міру їх отримання. Коли текст із механізмами прокрутки стає доступним користувачеві, браузер продовжує отримувати останні файли, необхідні для правильного відображення сторінки, зокрема картинки. Таким чином, щоб побачити сторінку, користувачу не потрібно чекати отримання всіх її компонентів (рис. 4.5), у результаті, web-браузер виконує декілька завдань одночасно. Розробка браузера у вигляді багатопотокового клієнта суттєво спрощує роботу з ним. Як тільки отримано основний файл HTML, активізуються окремі потоки виконання, які відповідають за дозавантаження інших частин сторінки. Кожен з потоків виконання створює окреме з'єднання із сервером і отримує від нього дані. Встановлення з'єднання й читання даних із сервера може бути запрограмоване

з використанням стандартних (блокуючих) системних викликів, які у цьому разі не можуть припинити основний процес завантаження сторінки. Код кожного з потоків виконання однаковий і переважно нескладний, у результаті чого користувач, хоч і помічає затримку в показі картинок, може переглядати документ.

Рис. 4.5. Поетапне завантаження елементів сторінки Web-браузером

Використання багатопотокових web-браузерів, які відкривають декілька з'єднань, дає також іншу перевагу. В попередньому прикладі були створені декілька з'єднань з одним і тим самим сервером. Якщо цей сервер сильно завантажений або просто повільний, то неможливо суттєво підвищити продуктивність порівняно з послідовним завантаженням файлів.

Проте здебільшого web-сервери можуть бути *реплікованими на декілька машин*, а кожен сервер міститиме однаковий набір web-документів. *Репліковані сервери* перебувають в одному місці й мають одне ім'я. Під час активації запиту на web-сторінці цей запит передається одному із серверів, зазвичай з використанням алгоритму циклічного обслуговування або іншого алгоритму вирівнювання навантаження.

За наявності *багатопотокового клієнта* з'єднання можуть бути встановленими з різними репліками, що потребує *паралельної передачі даних*, а це сприятиме тому, що web-документ буде показаний повністю значно швидше, ніж у разі нереплікованого сервера. Цей підхід спрацьовує, лише якщо клієнт має змогу обробити реально паралельні потоки вхідних даних, у цьому разі потоки виконання ідеально підходять для обробки потоків даних.

4.4.4 Багатопотокові сервери

Хоча багатопотокові клієнти мають істотні переваги, основні переваги багатопоточності в розподілених системах забезпечуються на стороні сервера. Практика показує, що багатопотоковість не лише істотно спрощує код сервера, але й робить набагато простішою розробку тих програмних серверів, у яких для досягнення високої продуктивності потрібне паралельне виконання декількох програмних комплексів, до яких належать і *мультипроцесорні системи*. Навіть нині, коли мультипроцесорні комп'ютери активно випускають у вигляді робочих станцій загального призначення, використання для паралельної обробки багатопотоковості не втратило своєї актуальності.

Приклад. Щоб відчути переваги потоків виконання для написання кодів серверів, розглянемо організацію *файлового сервера*, який періодично блокується, очікуючи доступу до диска. Файловий сервер зазвичай чекає вхідного запиту на операції з файлами, після чого обробляє отриманий запит і повертає відповідь. Одне з можливих архітектурних рішень показано на рис. 4.6, на якому один з потоків виконання (*диспетчер*) прочитує запити на файлові операції, які надсилають клієнти із зазначенням кінцевої точки цього сервера. Після перевірки запиту сервер обирає (тобто блокує) *робочий потік виконання*, який перебуває у стані очікування, і передає запит йому.

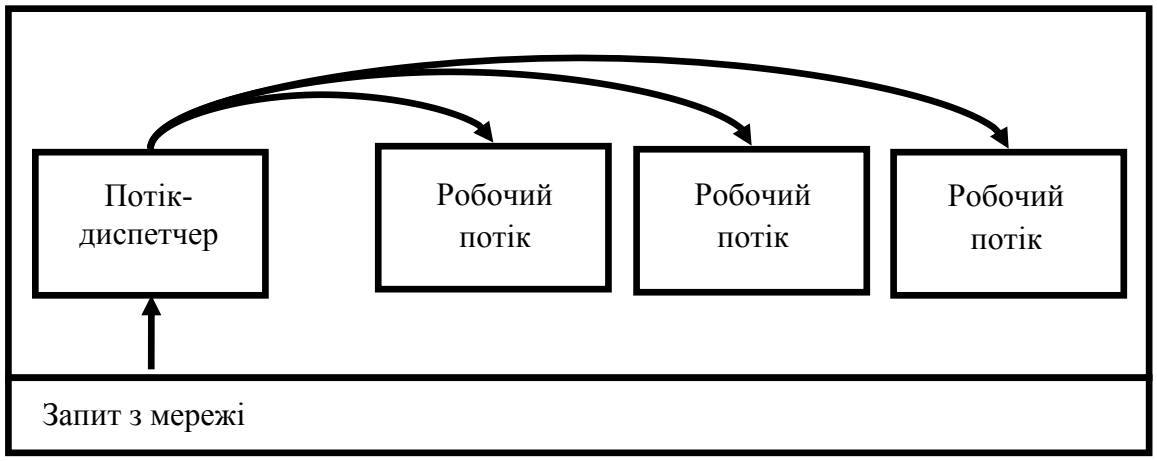


Рис. 4.6. Багатопотоковий сервер, організований
за схемою «диспетчер-робочий потік»

Робочий потік здійснює блокуюче читання з локальної файлової системи, що призупиняє потік виконання на час читання даних з диска. На час призупинення потоку виконання керування може бути передане іншому потоку виконання, напри-

клад потоку-диспетчеру, або диспетчер може обрати інший готовий до запуску робочий потік.

Структура з диспетчером – не єдиний спосіб організації багатопотокового сервера. У моделі «команда» всі потоки виконання еквівалентні, кожен отримує й обробляє власні запити (рис. 4.7). Іноді завдання надходять, а потрібний потік зайнятий, що зазвичай відбувається, коли кожен потік спеціалізується на виконанні особливого виду робіт. У цьому разі створюється *черга незавершених робіт*, за якої потоки виконання мають спочатку переглядати чергу робіт, а потім виконувати завдання, яке щойно надійшло.

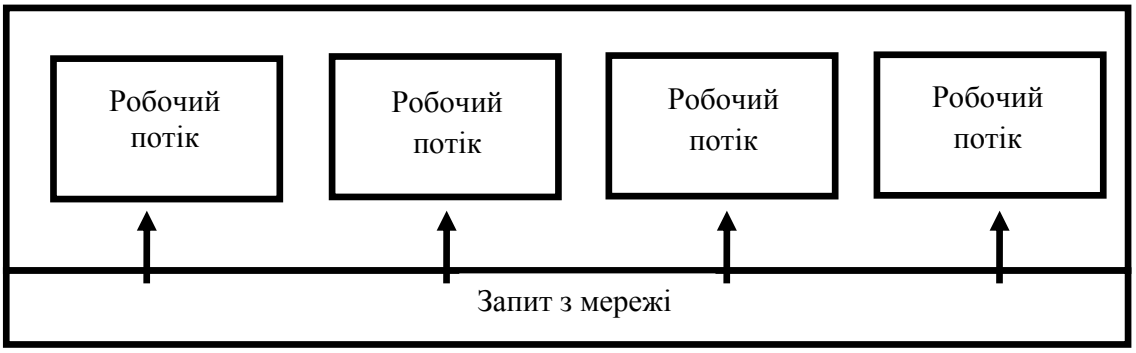


Рис. 4.7. Багатопотоковий сервер, який працює за схемою «команда»

Потоки виконання можуть також мати організацію розкладу обчислень у вигляді *конвеєра* (рис. 4.8), за якої перший потік породжує деякі дані й передає їх для обробки наступному потоку виконання і т. д. Незважаючи на те, що така організація і не підходить для файлового сервера, для інших завдань, зокрема задач «виробник-споживач», це рішення є ефективним.

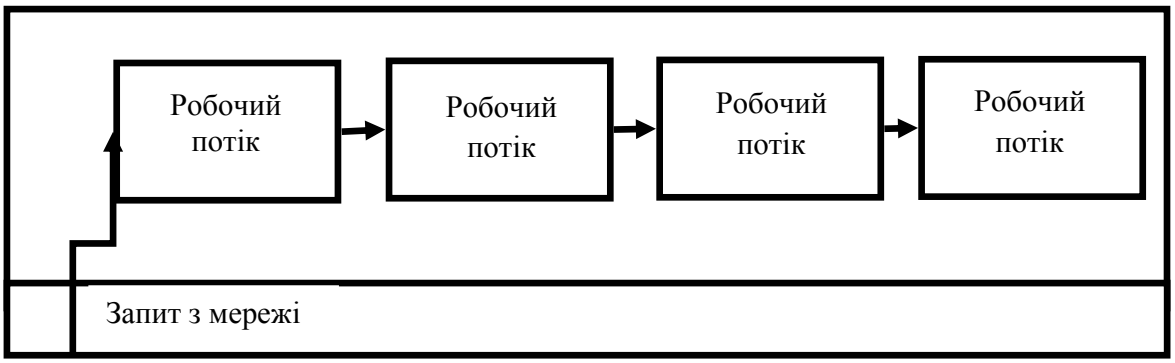


Рис. 4.8. Багатопотоковий сервер, який працює за схемою «конвеєр»

Розглянемо, як файловий сервер міг би виконувати запис, якщо немає потоків виконання. Одна з можливостей – виконувати обчислення так, ніби файловий сервер працює з єдиним потоком виконання. Основний потік файлового сервера отримує запит, перевіряє його і передає на виконання раніше, ніж отримає наступний. Поки сервер чекає закінчення дискових операцій, він не обробляє інших запитів від клієнтів. Окрім того, якщо файловий сервер працює на віддаленій машині, то процесор під час дискових операцій не зайнятий, у результаті чого обробляється значно менша кількість запитів за секунду. Таким чином, потоки виконання забезпечують суттєве підвищення продуктивності, незважаючи на те, що вони запускаються на виконання по черзі.

Наявні такі можливі організації обчислювального процесу на сервері: *багатопотоковий; однопотоковий сервер;* потоків виконання немає зовсім, але за умови, коли спадання продуктивності через використання однопотокової архітектури робить її застосування можливим. За іншої організації обчислювального процесу сервер використовують як великий кінцевий автомат (рис. 4.9).

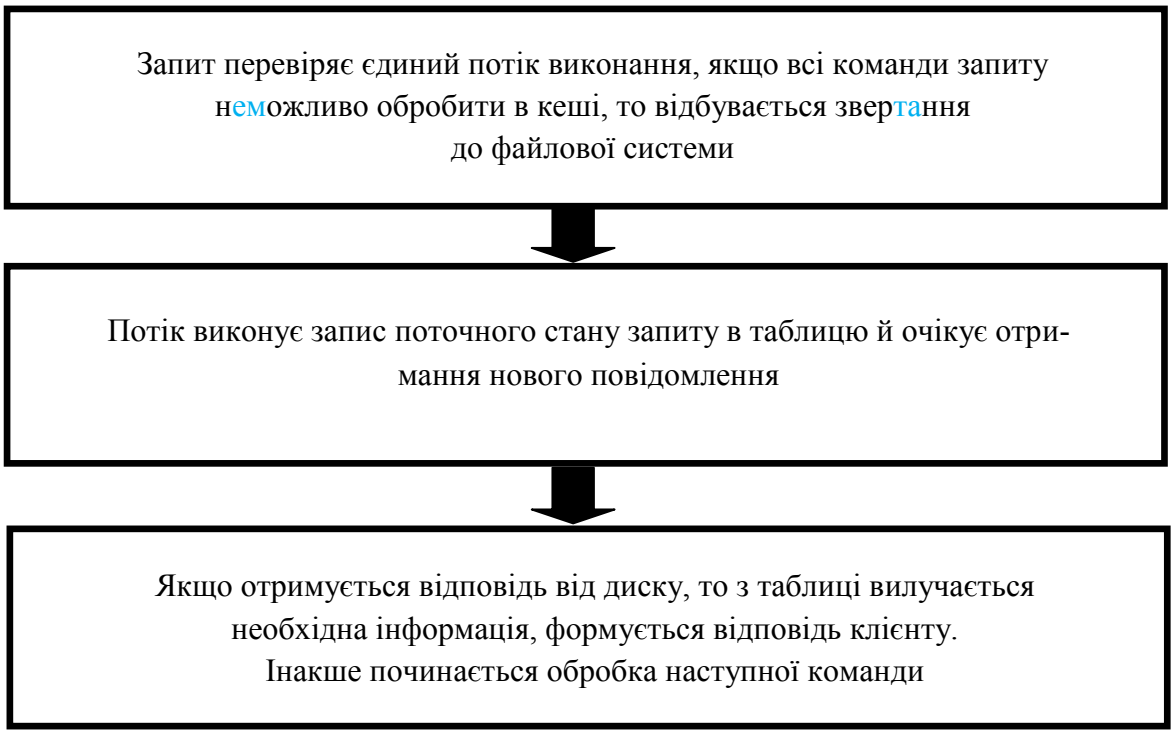


Рис. 4.9. Алгоритм роботи кінцевого автомата

Після надходження запиту його перевіряє єдиний потік виконання, якщо всі команди запиту неможливо обробити в кеші, то відбувається звертання до дискової системи. Проте замість блокування потік виконання записує стан поточного запиту в таблицю і переходить до стану очікування й отримання нового повідомлення. Нове повідомлення може бути як запитом на здійснення нової операції, так і відповіддю на попередній запит від дискової підсистеми. Якщо це новий запит, то починається нова робота, якщо це відповідь від диска, то з таблиці вилучається відповідна інформація, відповідь формується і передається клієнтові. Згідно з цією схемою сервер повинен мати можливість здійснювати неблокуючі виклики *send* і *receive*.

У такій архітектурі моделі «послідовних процесів» немає. Стан обчислень для кожного повідомлення, яке приймається і відправляється, має повністю зберігатися й отримуватися з таблиці, у результаті чого потоки виконання та їх стеки складно моделювати. Процес, який проходить у кінцевому автоматі, полягає у фіксації подій і, залежно від їх типу, реакції на ці події.

Потоки виконання уможливлюють одночасне збереження ідеї послідовних процесів, які здійснюють блокуючі системні виклики (як і під час викликів RPC для роботи з диском), і паралельної роботи. Блокуючі системні виклики спрощують програмування, а паралельність обробки потоків підвищує продуктивність. Однопотокові сервери зберігають простоту блокуючих системних викликів, але втрачають у продуктивності. Підхід, використовуваний у кінцевих автоматах, дозволяє завдяки паралелізму досягти високої продуктивності, але через наявність неблокуючих викликів важко програмується (табл. 4.1).

Таблиця 4.1. Способи побудови сервера

Модель	**Характеристики**
Потоки виконання	Паралельність обробки потоків, блокуючі системні виклики
Однопотоковий процес	Немає паралелізму, блокуючі системні виклики
Кінцевий автомат	Паралелізм, неблокуючі системні виклики

4.5 Клієнти

4.5.1 Інтерфейси користувача

Основне завдання більшості програмних клієнтів – бути передавальною ланкою між користувачем і віддаленим сервером, у такому разі їх основною функцією є підтримка інтерфейсу, призначеного для користувача. Здебільшого інтерфейс між користувачем та віддаленим сервером порівняно простий і вбудований в апаратуру клієнта.

Приклад. Стільникові телефони крім кнопок для набору номера мають невеликий дисплей. Робота з електронною поштою може вимагати комплектації справжньою клавіатурою, електронним планшетом або пристроєм для розпізнавання мови.

4.5.2 Клієнтське програмне забезпечення і прозорість розподілу

Програмне забезпечення клієнта охоплює призначений для користувача інтерфейс, і здебільшого на стороні клієнта виконується *частина рівнів обробки даних прикладної програми «клієнт-сервер»*. На основі вбудованого клієнтського програмного забезпечення функціонує клас спеціалізованих пристроїв, таких як автовідповідачі, лічильники купюр, зчитувачі штрих-коду, телеприставки та ін. У таких пристроях призначений для користувача інтерфейс є невеликою частиною клієнтського програмного забезпечення порівняно *із засобами локальної обробки* і *комунікацій.*

Крім того, клієнтське програмне забезпечення містить *компоненти,* які забезпечують *прозорість розподілу*. В ідеалі клієнт не має знати про свою взаємодію з віддаленими процесами, на відміну від серверів, де інформація про розподілену роботу зазвичай не приховується для підвищення продуктивності й коректності роботи. Очевидно, що сервери, між якими виконується реплікація, мають періодично зв’язуватися один з одним для забезпечення певного порядку виконання операцій у кожній з реплік.

Прозорість доступу зазвичай забезпечується за рахунок генерації коду програми-клієнта (у вигляді *заглушки клієнта*) з визначенням інтерфейсу тих функцій, які має виконувати сервер. *Заглушка* надає такий самий

інтерфейс, що і сервер, приховуючи при цьому *відмінності в архітектурі і реальній взаємодії*.

Розрізняють різні способи реалізації прозорості *розміщення*, *перенесення* та *переміщення*, для якої важливо використовувати певну *систему найменування.* Важливою відмінністю технології «клієнт–сервер» є кооперація з програмним забезпеченням клієнтської сторони. Якщо клієнт уже прив'язаний до сервера, то його потрібно повідомляти про зміну місцеперебування сервера. Проміжний рівень клієнта приховує дійсне місце розташування сервера від користувача й у разі потреби рівень непомітно повторить прив'язку до цього сервера. Найбільше, що може бути помічено у програмі клієнта, – це тимчасове спадання продуктивності.

Більшість розподілених систем реалізують прозорість реплікації на стороні клієнта. Розглянемо розподілену систему з віддаленими об'єктами, реплікацію яких можна здійснити через розсилання всім реплікам запиту, як показано на рис. 4.10. Програмний компонент на стороні сервера, який називають *замісником* клієнта, має зібрати всі відповіді й передати прикладній програмі клієнта одне *повертане значення.*

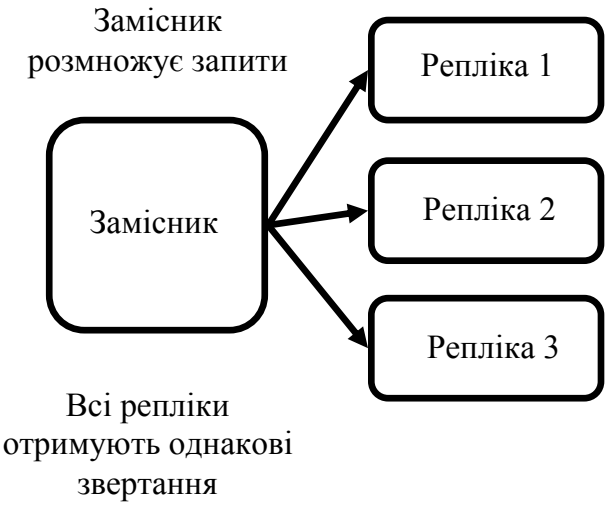


Рис. 4.10. Можливий підхід до прозорої реплікації віддалених об'єктів з використанням клієнтського програмного забезпечення

Розглянемо прозорість до збоїв. Маскування збоїв у взаємодії з серверами зазвичай виконується за допомогою клієнтського програмного забезпечення проміжного рівня, яке можна конфігурувати так, щоб воно багаторазово на-

магалося зв’язатися із сервером або обирало після декількох спроб інший сервер. Можлива також ситуація, коли програмне забезпечення клієнта, якщо web-браузер не може зв’язатися із сервером, повертає дані, збережені в кеші під час попереднього сеансу зв’язку.

Прозорість паралельного виконання може забезпечуватися *спеціальними проміжними серверами*, або *моніторами транзакцій*, і потребує меншої підтримки з боку клієнтського програмного забезпечення. *Прозорість збереження* також реалізується серверами.

4.6 Сервери

4.6.1 Підходи до побудови серверів прикладного програмного забезпечення

Сервер – це процес, який реалізує деяку службу, потрібну групі клієнтів. Усі сервери працюють за схожими алгоритмами: вони очікують на появу вхідного повідомлення, яке надсилає клієнт, перевіряють його на правильність, після чого чекають появи наступного повідомлення.

Сервери можуть бути організовані різними способами. Якщо *сервер ітеративний*, то він сам обробляє запит й у разі потреби повертає клієнтові у відповідь повідомлення, тоді як *паралельний сервер* не обробляє повідомлення сам, а передає його в окремий потік виконання або інший процес, після чого відразу переходить у стан очікування наступного вхідного повідомлення.

Приклад. Розглянемо *багатопотоковий сервер.* У реалізації такого паралельного сервера на кожен запит, що надійшов, може створюватися новий процес. Такий підхід застосовується в багатьох UNIX-системах. Потік або процес, який обробляє запит, відповідає за відправлення відповіді клієнтові, що надіслав запит.

Клієнт завжди надсилає запити в *кінцеву точку (endpoint)*, яку називають **портом** (port), тієї машини, на якій працює сервер. Кожен сервер переглядає вказану йому кінцеву точку. Одне з рішень проблеми щодо визначення кінцевих точок клієнтом, яку складно розв’язати у загальному випадку, є глобальне призначення кінцевих точок найбільш поширеним службам.

Приклад. Сервери, які обслуговують запити до FTP в Internet, завжди працюють з портом 21, HTTP-серверам World Wide Web завжди призначають TCP-порт 80. Ці кінцеві

точки визначені організацією реєстрації номерів Internet (Internet Assigned Numbers Authority – IANA). Маючи визначені кінцеві точки, клієнтові достатньо знати мережну адресу тієї машини, на якій запущена служба. Для цього використовують служби іменування.

Багато служб не потребують попереднього призначення кінцевої точки.

Приклад. Служба часу може використовувати кінцеву точку, що в динаміці виглядає для неї як локальна операційна система. У такому разі клієнт має спочатку визначити кінцеву точку.

Для визначення кінцевої точки клієнтом у процесі виконання розподіленої прикладної програми розроблено рішення, яке реалізовано *у розподіленому середовищі Distributed Computing Environment (DCE),* яким є створення спеціального *демона*, що запускається на кожній машині, на якій працюють сервери. Демон відстежує поточну кінцеву точку кожного сервера, що реалізується на ньому, переглядає загальнодоступні кінцеві точки. Під час першого контакту з демоном клієнт запитує кінцеву точку, після чого зв’язується з потрібним сервером. Зазвичай кінцева точка асоціюється з конкретною службою, проте на практиці реалізація кожної служби окремим сервером є недоцільним витрачанням ресурсів.

Приклад. У типових UNIX-системах зазвичай працює одночасно багато серверів, при цьому велика їх частина пасивно чекає запитів від клієнта. Замість ведення такої кількості пасивних процесів часто ефективніше мати один суперсервер (superserver), який прослуховує всі кінцеві точки, що стосуються конкретних служб, як показано на рис. 4.11. Такий підхід реалізує, наприклад, демон *inetd* у UNIX, який переглядає стандартні порти служб Internet і, якщо надійшов запит, то розпаралелює процес і передає запит для подальшої обробки, після закінчення якої дочірній процес завершується.

Приклад. Нехай користувач вирішив завантажити на FTP-сервер великий файл. Почавши робити це, він зрозумів, що обрав не той файл, і хотів би відмінити подальшу пересилку даних. Використовують декілька способів це зробити. Один з підходів, що надійно працює в сучасному Internet (інколи це єдиний можливий підхід), – користувач негайно закриває клієнтське програмне забезпечення, що автоматично викликає процес розриву з’єднання із сервером, потім перезапускає його і продовжує роботу. Сервер розриває старе з’єднання, вважаючи, що клієнт припинив роботу.

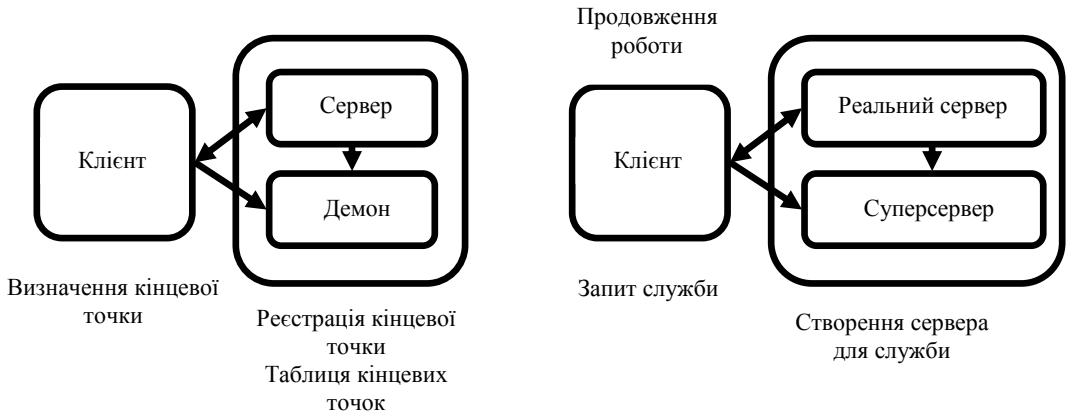


Рис. 4.11. Прив'язка клієнта до сервера з використанням демона в середовищі ОС і суперсервера в UNIX

Найбільш коректний спосіб викликати переривання зв'язку – розробляти програмне забезпечення клієнта і сервера так, щоб вони могли пересилати один одному інформацію *(out-ofband)* стосовно сервера, який має обробляти раніше за решту даних, які передаються клієнтом. Один з підходів до застосування сигналу кінця зв'язку – змусити сервер переглядати окрему кінцеву точку, на яку клієнт його відправлятиме й одночасно (з нижчим пріоритетом) переглядати кінцеву точку, через яку передаються нормальні дані. Інший підхід – пересилати сигнал кінця зв'язку тим самим з'єднанням, через яке клієнт пересилав свій запит. У протоколі TCP, наприклад, можна надсилати термінові дані. Коли термінові дані досягають сервера, він перериває свою роботу (в UNIX-системах – за сигналом), після чого може проглянути ці дані й обробити їх.

Важливою характеристикою сервера є можливість збереження інформації про стан клієнтів на сервері. *Сервер без фіксації стану (stateless sewer)* не зберігає інформацію про стан своїх клієнтів і може змінювати власний стан, не інформуючи про це клієнтів.

Приклад. Web-сервер – це сервер без фіксації стану, який відповідає на вхідні HTTP-запити, які можуть вимагати завантаження файлу як на сервер, так і (набагато частіше) з сервера. Після виконання запиту Web-сервер забуває про клієнта. Набір файлів, якими керує Web-сервер (можливо, в комбінації з файловим сервером), може бути змінений без повідомлення клієнтів про цю дію.

На відміну від сервера без фіксації стану, *сервер із фіксацією стану* (*stateful server*) зберігає та обробляє інформацію про своїх клієнтів.

Приклад. Розглянемо файловий сервер, що дозволяє клієнтові створювати локальні копії файлів, скажімо, для підвищення продуктивності операцій оновлення. Такий сервер підтримує таблицю, яка містить записи пар (клієнт, файл) та дозволяє відстежувати, який клієнт з яким файлом працює, завдяки чому сервер завжди визначає «найсвіжішу» версію файлу. Цей підхід підвищує продуктивність операцій «читання–запису», які здійснюються на клієнтській стороні.

Підвищення продуктивності порівняно із серверами без фіксації стану найчастіше є основними перевагою і причиною розробки серверів із фіксацією стану. Розглянутий приклад ілюструє й основний недолік таких серверів. У разі збою сервера він вимушений відновлювати свою таблицю із записами пар (клієнт, файл), інакше не буде жодної гарантії в тому, що робота виконується з останньою оновленою версією файлу. Зазвичай сервери з фіксацією стану потребують відновлення свого стану в тому вигляді, в якому вони були до збою, що дуже ускладнює програмне забезпечення. У разі архітектури без фіксації стану взагалі немає потреби вживати якихось спеціальних заходів для відновлення серверів після збою. Вони просто перезапускаються і працюють, чекаючи запитів клієнта.

У процесі розробки сервера вибір між архітектурою з фіксацією і без фіксації стану не відображається на службі, яку цей сервер має надавати. Так, оскільки до виконання операцій читання або запису файли потрібно відкривати, сервер без фіксації стану має виконати відкриття файлу. Стандартний підхід полягає в тому, щоб сервер, обробляючи запит на запис у файл або читання з файлу, відкривав потрібний файл, виконував операцію читання чи запису і негайно його закривав.

В інших випадках сервер може зберігати записи про поведінку клієнтів, щоб ефективніше обробляти їх запити, але лише за умови, що сервер зберігатиме інформацію про клієнта. Традиційне рішення полягає в тому, щоб дозволити клієнтові надсилати додаткову інформацію про його попередні сеанси роботи із сервером, яку часто прозоро зберігає браузер клієнта у вигляді *файлів* ***cookie*** – маленьких фрагментів даних, що містять інформацію про клієнта,

важливу для сервера. Файли *cookie* ніколи не виконуються браузером, а просто зберігаються.

Під час першого звертання клієнта до сервера останній пересилає файл *cookie* разом із запитаними Web-сторінками браузеру, а браузер зберігає цей файл *cookie*. Кожного разу під час звертання клієнта до сервера файли *cookie* пересилаються браузером на сервер разом із текстом запиту. Теоретично цей підхід працює добре, на практиці ж файли *cookie* для безпечного зберігання відсилаються браузером назад на сервер так, щоб цей процес був скритим від користувача.

4.6.2 Сервери об'єктів

Сервер об'єктів (object server) – *це сервер, орієнтований на підтримку розподілених об'єктів.*

Важлива відмінність між стандартним сервером об'єктів та іншими серверами полягає в тому, що сам сервер об'єктів не надає конкретної служби, а лише засоби звертання до локальних об'єктів на основі *запитів від віддалених клієнтів*. *Конкретні служби* реалізують об'єкти, розміщені на сервері, що дозволяє порівняно легко змінити набір служб, додаючи або видаляючи об'єкти. *Сервер об'єктів*, відповідно, відіграє роль місця для зберігання об'єктів.

Об'єкт складається з двох частин: *даних, які відображають його стан, і коду, що створює реалізацію його методів* (рис. 4.12). Чи будуть ці частини зберігатися окремо, а також чи зможуть методи спільно використовуватися декількома об'єктами, залежить від сервера об'єктів. Спосіб звертання сервера об'єктів до тих об'єктів, які на ньому зберігаються, має певні особливості.

Приклад. У багатопотоковому сервері об'єктів окремий потік виконання може бути призначений окремому об'єкту або запиту на звертання до об'єкта.

Для будь-якого об'єкта, до якого виконується звертання, сервер об'єктів має знати, який код виконувати, з якими даними працювати, чи запускати окремий потік виконання для підтримки звертання тощо. Якщо вважати, що всі об'єкти виглядають однаково, то звертання до них можна утворювати однаково. Саме так працює середовище DCE. На жаль, такому підходу зазви-

чай не вистачає гнучкості, через що він має певні обмеження у процесі роботи з розподіленими об’єктами.

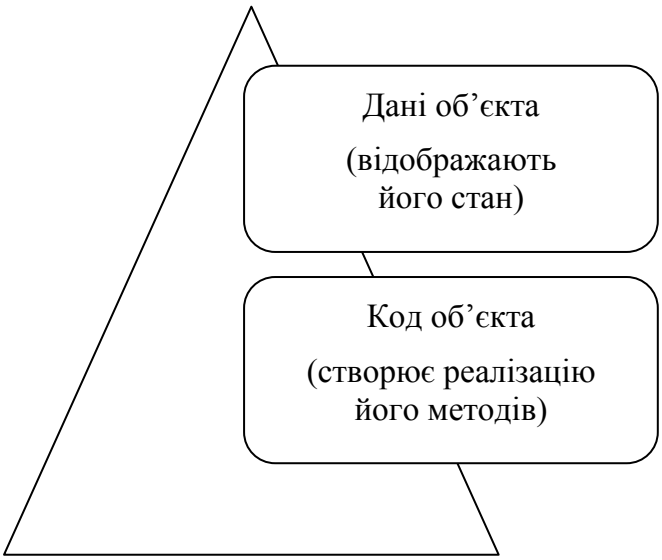


Рис. 4.12. Структура сервера об’єктів

Найбільш логічний підхід з боку сервера – підтримувати різні правила обробки об’єктів. Розглянемо, як приклад, роботу з *нерезидентними об’єктами*. **Нерезидентним** називають об’єкт, який існує лише під час існування сервера, а можливо, і ще коротший термін. Типовою реалізацією нерезидентного об’єкта можна вважати копію файлу, яка зберігається в пам’яті, призначену лише для читання, наприклад калькулятор (який зазвичай запускається на високошвидкісному сервері). Доцільно створювати нерезидентний об’єкт під час першого запиту до нього, а знищувати після того, як не залишиться пов’язаних із цим об’єктом клієнтів.

Перевага такого підходу полягає в тому, що нерезидентні об’єкти використовують ресурси сервера лише доти, доки в цих об’єктах є потреба; недолік – у тому, що звертання потребує для виконання певний додатковий час на створення об’єкта. З огляду на це інколи застосовують інший підхід – всі нерезидентні об’єкти створюють у процесі ініціалізації сервера, виділяючи ресурси на об’єкти навіть у тому разі, якщо жоден клієнт не стане їх використовувати.

Аналогічно сервер має можливість виділяти для кожного зі своїх об’єктів окремий сегмент пам’яті, тобто заборонити спільне використання

об’єктами коду і даних. Такий підхід застосовують тоді, коли для реалізації кожного об’єкта не потрібно виокремлювати код від даних або коли об’єкти потрібно відокремити один від одного з міркувань безпеки, тобто сервер для гарантованого захисту меж сегментів має надати спеціальні засоби вимірювання часу або вимагати підтримки від базової операційної системи. У разі альтернативного підходу дозволяють об’єктам спільно використовувати код. База даних, що містить об’єкти, які належать до певного класу, може бути ефективно реалізована за рахунок однократного завантаження на сервер реалізації цього класу. В разі надходження запиту на звертання до об’єкта серверу достатньо отримати з бази даних стан об’єкта і виконати викликаний метод.

Аналогічно розрізняють інші підходи до роботи з потоками виконання. Найпростіший з них – реалізація сервера з єдиним керівним потоком виконання. Сервер може підтримувати і декілька потоків виконання – по одному на кожен об’єкт. У разі надходження запиту зі звертанням до об’єкта сервер передає запит потоку виконання, який відповідає за цей об’єкт. Якщо потік виконання у цей момент зайнятий, то запит стає в чергу. Перевага такого підходу полягає в автоматичному захисті об’єктів від одночасного доступу. Для єдиного пов’язаного з об’єктом потоку виконання всі звертання вишиковуються по черзі. Можна також виділяти окремий потік виконання кожному запиту, але необхідно заздалегідь запобігти одночасному доступу до об’єктів. Незалежно від того, чи призначається потік виконання кожному об’єкту або кожному методу, потрібно вирішити, чи створювати кожен потік за запитом, чи підтримувати на сервері пул потоків. Універсального оптимального рішення цієї проблеми бути не може.

4.7 Перенесення коду

У попередніх розділах розглянуто розподілені системи, в яких взаємодія обмежувалася *передачею даних*, проте є ситуації, коли *передача програм*, інколи навіть під час їх виконання, дозволяє спростити розробку розподілених систем. Розглянемо таку складну проблему, як *перенесення коду* в гете-

рогенних системах. Найбільш поширеною реалізацією є система D'Agents для мобільних агентів.

4.7.1 Підходи до перенесення коду

Традиційно *перенесення коду* в розподілених системах відбувається у формі *перенесення процесів (process migration)*, у разі якого процес цілком переноситься з однієї машини на другу. Перенесення працюючого процесу на іншу машину – дороге і складне завдання, і для його виконання має бути вагома причина, зокрема продуктивність, яка може підвищуватися, якщо процеси перенесено із сильно завантаженої на слабо завантажену машину. **Завантаження** зазвичай виражається в поняттях «*довжина черги до процесора*» або «*завантаження процесора»,* а також інших індикаторах продуктивності.

Алгоритм розподілу завантаження, на основі якого приймаються рішення та який охоплює розподіл і перерозподіл завдань відповідно до наявного набору процесорів, відіграє важливу роль у системах інтенсивних обчислень. Разом з тим, у багатьох сучасних розподілених системах *оптимізація обчислювальної потужності* – менш важливе завдання порівняно, наприклад, зі *зниженням комунікаційного трафіку.* Враховуючи гетерогенність базових платформ і комп'ютерних мереж, підвищення продуктивності за рахунок перенесення коду іноді ґрунтується значною мірою на показниках якості обслуговування, а не на математичних моделях.

Приклад. Розглянемо систему «клієнт–сервер», у якій сервер керує великою базою даних. Якщо клієнтське програмне забезпечення має виконувати низку операцій з базою даних, використовуючи великі обсяги даних, то краще перенести частину клієнтського програмного забезпечення на сервер, а мережею передавати лише результати. Інакше мережа може бути переобтяжена даними, які передаються із сервера на клієнтську машину. В такому разі перенесення коду виконується через те, що зазвичай дані обробляють поблизу того місця, де вони містяться.

Схожа ситуація спостерігається й у разі перенесення частини сервера на комп'ютер клієнта. У багатьох інтерактивних прикладних програмах баз даних клієнт має заповнювати форму, яка потім буде перетворена в серію операцій бази даних. Обробка форми на стороні клієнта з пересиланням на сервер лише заповненої форми дозволяє запобігти пересиланню мережею значної кількості

невеликих повідомлень, у результаті чого клієнт підвищить продуктивність, а сервер витратить менше часу на обробку форми і процеси взаємодії.

Підтримка перенесення коду може також допомогти підвищити продуктивність за рахунок використання паралелізму, але без труднощів, пов'язаних із паралельним програмуванням.

Приклад. Розглянемо пошук інформації у Web-середовищі. Порівняно просто реалізувати пошуковий запит у вигляді невеликої мобільної програми, що переноситься із сайту на сайт. Створивши декілька копій цієї програми і розіславши їх на різні сайти, можна досягти лінійного зростання швидкості пошуку порівняно з єдиним екземпляром програми.

Окрім підвищення продуктивності, наявні й інші причини підтримки перенесення коду, найбільш важливою з яких є *гнучкість*. Традиційний підхід до побудови розподілених програмних засобів полягає в розбитті прикладної програми на частини з подальшим визначенням, де яка частина виконуватиметься. Проте якщо код переносити з машини на машину, то з'являється можливість конфігурувати розподілені системи динамічно.

Приклад. Розглянемо сервер, який реалізує стандартизований інтерфейс до файлової системи. Щоб надати віддаленому клієнтові доступ до файлової системи, цей сервер використовує спеціальний протокол. Зазвичай клієнтська реалізація інтерфейсу з файловою системою, відповідно цього протоколу, має бути скомпонована з прикладною програмою клієнта. Цей підхід передбачає, що програмне забезпечення такого інтерфейсу клієнта має бути доступним для нього вже тоді, коли створюється клієнтське програмне забезпечення.

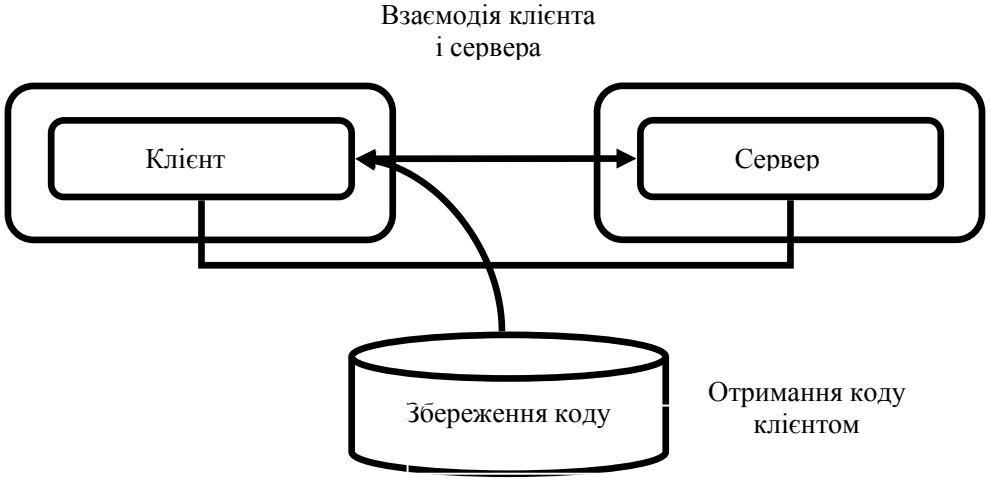


Рис. 4.13. Принцип динамічної конфігурації клієнта для зв'язку із сервером

Важлива перевага такої моделі динамічного дозавантаження клієнтського програмного забезпечення полягає в тому, що клієнтові для спілкування з сервером немає потреби у повному комплекті заздалегідь установлюваного програмного забезпечення. Замість цього програми можуть бути перенесені на клієнта і так само видалені, коли потреба в них зникне. Інша перевага – оскільки інтерфейси стандартизовані, то можна змінювати протокол взаємодії «клієнт–сервер» і його реалізацію за потребою. Зміни не вплинуть на наявні клієнтські програмні засоби, пов'язані з сервером.

Недолік такого підходу – код, який завантажується, не реалізує лише оголошених інтерфейсів доступу до незахищеного жорсткого диска і може відправляти певні фрагменти інформації невідомому споживачу.

4.7.2 Моделі перенесення коду

Хоча *перенесення коду* передбачає лише *переміщення коду з машини на машину* цей термін має більш розширену сферу застосування. Традиційно зв'язок у розподілених системах ґрунтується на обміні даними між процесами. *Перенесення коду* в широкому значенні пов'язане з *перенесенням програм* з машини на машину для виконання цих програм у потрібному місці. У деяких випадках, таких як перенесення процесів, також потрібно переносити стан програми, отримувані сигнали та інші елементи середовища.

Для кращого розуміння різних моделей перенесення коду розглянемо *шаблон*, згідно з яким процес складається *з трьох сегментів*. **Сегмент коду** – це частина, що містить набір інструкцій, які виконуються у процесі виконання програми. **Сегмент ресурсів** містить посилання на зовнішні ресурси, необхідні процесу, зокрема на файли, принтери, пристрої, інші процеси. **Сегмент виконання** використовується для зберігання поточного стану процесу, включаючи закриті дані, стек і лічильник програми.

Абсолютний мінімум для перенесення коду пропонує модель **слабкої мобільності** (weak mobility), згідно з якою допускається перенесення лише *сегмента коду*, можливо, разом з деякими *даними ініціалізації*. Характерною рисою слабкої мобільності є те, що перенесена програма завжди за-

пускається зі свого вихідного стану. Це відбувається, наприклад, якщо використовувати Java-аплети. Перевага такого підходу полягає в його простоті. Слабка мобільність потрібна лише для того, щоб машина, на яку переноситься код, мала змогу його виконувати, чого цілком достатньо, щоб зробити код переносним.

На відміну від слабкої мобільності, в системах, що підтримують **сильну мобільність** *(strong mobility)*, переноситься також *сегмент виконання*. Характерна риса сильної мобільності – працюючий процес може бути припинений, перенесений на другу машину і його виконання продовжено з того місця, на якому його припинено. Зрозуміло, що сильна мобільність значно потужніша від слабкої, але і значно складніша в реалізації. Прикладом системи, що підтримує сильну мобільність, є система D'Agents.

Незалежно від того, є мобільність слабкою або сильною, слід розрізняти *системи з перенесенням, ініційованим відправником*, і *системи з перенесенням, ініційованим одержувачем*. Якщо перенесення ініційовано відправником, то це означає, що перенесення ініціюється машиною, на якій код, що переноситься, постійно розміщений або виконується. Зазвичай перенесення, яке ініціює відправник, відбувається під час завантаження програм на обчислювальний сервер.

Приклад. Передача пошукових програм через Internet на сервер баз даних у Web-браузер для виконання запиту на цьому сервері.

У разі перенесення ініційованого одержувачем, ініціатива щодо перенесення коду належить машині-одержувачеві. Прикладом такого перенесення коду є Java-аплети.

Перенесення, ініційоване одержувачем, зазвичай реалізується простіше. Здебільшого перенесення коду відбувається між клієнтом і сервером, причому ініціативу бере на себе клієнт. Безпечне перенесення коду на сервер, як це відбувається у разі перенесення, ініційованого відправником, часто передбачає, щоб клієнт був спочатку зареєстрований і впізнавався сервером, тобто сервер має знати всіх своїх клієнтів, оскільки вони мають доступ до ресурсів сервера (наприклад, до диска), захист яких має обов'язково бути організованим. На противагу цьому, завантаження коду в разі ініціації про-

цесу приймальною стороною може здійснюватися анонімно. Сервер зазвичай не зацікавлений у ресурсах клієнта, а перенесення коду на клієнта виконується виключно з метою збільшення продуктивності клієнта. У такому разі в захисті має потребу лише невелика кількість ресурсів, таких як пам'ять і мережне з'єднання.

У разі слабкої мобільності слід також розрізняти варіанти: чи *перенесений код виконується у процесі-приймачі*, чи він *виконується у новому, спеціально запущеному процесі*.

Приклад. Java-аплети просто завантажуються у web-браузер і виконуються в адресному просторі браузера.

Перевага такого підходу полягає в тому, що немає потреби запускати новий процес і розривати через це зв'язок з машиною-приймачем. Основний *недолік* полягає в тому, що процес-приймач доводиться захищати від навмисного або випадкового виконання коду, для чого операційна система створює окремий процес для переміщеного коду, але це не вирішує проблем з доступом до ресурсів.

Окрім перенесення функціонуючого процесу, або *міграції процесу*, сильна мобільність може також здійснюватися за рахунок *віддаленого клонування*. На відміну від міграції процесу, **клонування** створює точну копію (**клон**) вихідного процесу, яка виконується на віддаленій машині паралельно оригіналові. У UNIX-системах віддалене клонування відбувається під час *відгалуження дочірнього процесу* в тому разі, коли цей дочірній процес продовжує виконання на віддаленій машині. Перевага клонування – у схожості зі стандартними процедурами, здійснюваними в численних програмних засобах. Єдина відмінність між ними полягає в тому, що клонований процес виконується на другій машині, тому міграція за допомогою клонування – найпростіший спосіб підвищення прозорості розподілу. Різні варіанти перенесення коду ілюструє рис. 4.14.

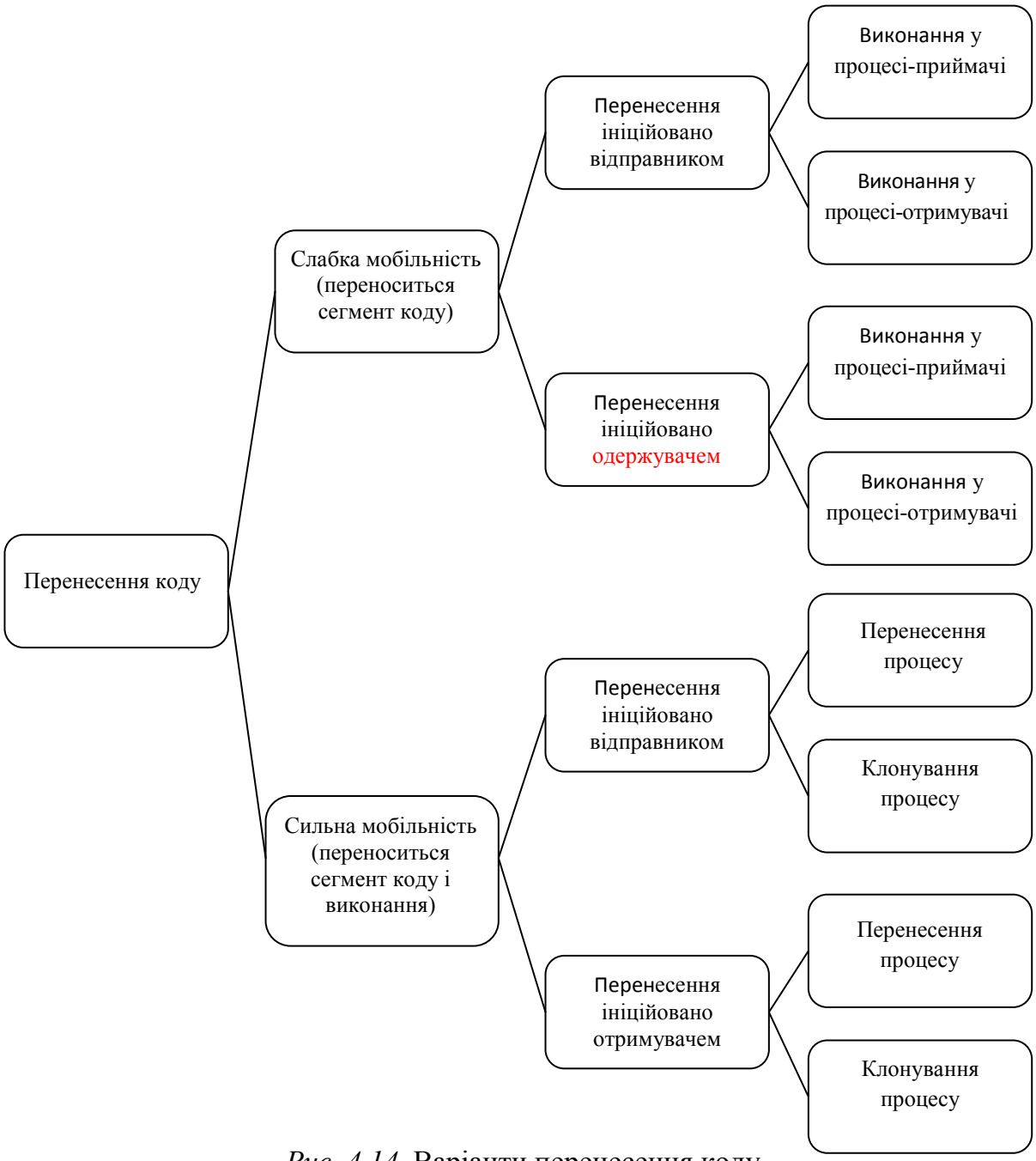


Рис. 4.14. Варіанти перенесення коду

4.7.3 Перенесення і локальні ресурси

У розділі 4.7.2 розглянуто перенесення лише сегментів коду і виконання, перенесення **сегмента ресурсів** є значно складнішим через потребу одночасного виконання процесів перенесення коду, виконання і ресурсів, а також че-

рез те, що ресурси не завжди можна перенести так само легко без змін, як інші сегменти.

Приклад. Розглянемо процес, який містить посилання на конкретний порт TCP, за допомогою якого він взаємодіє з іншими (віддаленими) процесами. Це посилання міститься в сегменті ресурсів процесу, який у разі перенесення процесу на іншу машину має звільнити зайнятий ним порт і запитати інший – на тій машині, на яку він був переміщений, що створює проблему коректної адресації. У деяких випадках перенесення посилання проблем не створює. Так, посилання на файл з використанням абсолютної URL-адреси залишиться правильним незалежно від того, на якій машині виконується процес, який містить цю URL-адресу.

Щоб зрозуміти, як впливає перенесення коду на сегмент ресурсів, було виокремлено *три типи зв'язків процесу з ресурсами*. Найбільш **сильний зв'язок** спостерігається, коли процес посилається на ресурс *за його ідентифікатором* і потребує саме того ресурсу, на який посилається.

Приклад. Така прив'язка за ідентифікатором *(binding by identifier)* є використанням процесом URL-адреси для посилання на конкретний web-сайт або Internet–адреси для посилання на FTP-сервер. Аналогічно, посилання на локальну кінцеву точку взаємодії також вважатиметься прив'язкою за ідентифікатором.

Слабший зв'язок процесу з ресурсами наявний у тому разі, коли процесу необхідне лише значення ресурсу, тоді виконання процесу зовсім не зміниться, якщо таке саме значення йому надасть інший ресурс.

Приклад. Типовою прив'язкою за значенням *(binding by value)* є звертання програм до стандартних бібліотек, як у разі програмування мовою C або Java, які завжди доступні на локальній машині, але їх дійсні місця розташування в локальній файловій системі можуть бути різними. Для правильного виконання процесу важливі не конкретні імена файлів, а їх вміст.

Найбільш слабка форма зв'язку відбувається тоді, коли процес потребує використання ресурсу певного типу.

Приклад. Прив'язка за типом *(binding by type)* може бути проілюстрована посиланнями на локальні пристрої, зокрема принтери, монітори.

Під час перенесення коду часто необхідно змінювати посилання на ресурси, не змінюючи типу прив'язки ресурсу до процесу. Проблемним питанням є таке: чи можна змінювати ресурси, і якщо так, то як це залежить від того, чи можуть вони бути перенесені на машину-приймач разом з кодом. Фактично, необхідно визначити *зв'язок ресурсів з машиною* і розглянути

низку варіантів організації такого зв'язку. *Неприєднанні ресурси (unattached resources)* можуть бути легко перенесені з машини на машину, *файли (даних)* у цьому разі зазвичай пов'язані лише з програмою, що переноситься. Перенесення або копіювання *зв'язаних ресурсів (fastened resources)* можливе лише з порівняно великими витратами. Типовими прикладами зв'язаних ресурсів можуть бути локальні бази даних або Web-сайти. Незважаючи на те, що ці ресурси теоретично не залежать від поточної машини, часто буває неможливо перенести їх в інше середовище.

Фіксовані ресурси (fixed resources) прив'язані до конкретної машини або середовища і не можуть бути перенесені на іншу. *Фіксованими ресурсами* часто бувають локальні пристрої, локальні кінцеві точки взаємодії тощо.

Скомбінувавши три типи прив'язки ресурсів до процесів і три типи прив'язки ресурсів до машини, отримаємо дев'ять комбінацій (варіантів перенесення ресурсів).

Розглянемо спочатку можливості, що виникають у разі прив'язки процесу до ресурсу за ідентифікатором. Якщо ресурс не приєднаний, то найкраще перенести його на іншу машину разом з кодом, проте якщо цей ресурс використовується переносним процесом спільно з іншими, то слід організувати на нього глобальне посилання – посилання, яке зможе подолати межу між машинами. Прикладом такого посилання може бути URL. Якщо ресурс зв'язаний або фіксований, то організація глобального посилання також може бути використана. Реалізація системи глобальних посилань може бути складнішою від простого використання URL й інколи виявляється дуже дорогою.

Приклад. Розглянемо програму обробки високоякісних зображень на окремій робочій станції. Створення в реальному часі високоякісних зображень – це завдання, яке вимагає інтенсивних обчислень, тому програма може бути перенесена на високопродуктивний обчислювальний сервер. Організація глобальних посилань на робочу станцію означатиме організацію зв'язку між сервером і робочою станцією. Правильна обробка, що відбувається одночасно на сервері й робочій станції, передбачає дотримання певних вимог до швидкості передачі зображень. У результаті чого може виявитися, що перенесення програми на обчислювальний сервер є невиправданим через високу вартість підтримки глобальних посилань.

Виникають проблеми з підтримкою глобальних посилань під час перенесення процесу, який використовує локальну кінцеву точку взаємодії. У цьому разі йдеться про фіксовані ресурси, прив'язані до процесу за ідентифікатором, тому є два рішення.

Перше – дозволити процесу після перенесення встановити з'єднання з вихідною машиною, створивши там окремий потік виконання, який просто буде перенаправляти всі повідомлення, які надходять, на нове місце перебування процесу. Основним недоліком цього підходу є те, що у разі збоїв або пошкодження вихідної машини зв'язок з перенесеним процесом буде перерваний.

Друге рішення полягає в тому, щоб у всіх процесах, пов'язаних із перенесенням, змінити їх глобальні посилання і пересилати повідомлення на нову кінцеву точку взаємодії цільової машини.

Процес прив'язки відбувається по-іншому в разі *прив'язки за значенням*. Розглянемо спочатку фіксовані ресурси. Комбінація *фіксованих ресурсів і прив'язки за значенням* можлива, наприклад, у разі використання процесом ділянки пам'яті спільно з іншими процесами. Організація глобальних посилань у цьому разі потребуватиме реалізації розподіленої розділюваної пам'яті, проте найчастіше подібне рішення неприйнятне.

Зв'язані ресурси, посилання на яких виконується за значенням, – це зазвичай бібліотеки часу виконання. Допускається копіювання цих ресурсів на іншу машину, яке можна здійснити до перенесення коду. Організація глобальних посилань може виявитися хорошою альтернативою копіюванню в тому разі, якщо потрібно скопіювати великий обсяг даних, наприклад словники текстового редактора.

Найбільш простий випадок – *неприєднані ресурси*. Найкраще рішення у цьому разі – скопіювати (або перемістити) ресурси в нове місце, уникаючи варіантів, коли вони спільно використовуються декількома процесами, коли єдино можливим є створення глобальних посилань.

Останній варіант – *прив'язка за типом*. Незалежно від способу прив'язки ресурсу до машини рішення полягає *в новій прив'язці процесу до локальних ресурсів* того самого типу, лише за умови, якщо необхідного типу ресурсів на

локальній машині немає, можна скопіювати або перемістити оригінальні ресурси на нове місце або організувати глобальні посилання на них.

4.8 Перенесення коду в гетерогенних системах

Для гомогенних систем передбачено припущення, що перенесений код може бути легко виконаний на цільовій машині, але розподілені системи створюються з набору гетерогенних платформ, кожна з яких має власну машинну архітектуру й операційну систему. Перенесення в таких розподілених системах передбачає, щоб підтримувалися всі платформи, на яких функціонує система, і сегмент коду має виконуватися на всіх платформах без перекомпіляції тексту програми, а сегмент виконання на кожній платформі має бути правильно наданим.

Проблеми можна частково усунути, якщо обмежитися *слабкою мобільністю*, коли зазвичай немає інформації щодо часу виконання, яку треба передавати від машини до машини. Це означає, що достатньо скомпілювати вихідний текст програми, створивши різні варіанти сегмента коду – по одному на кожну потенційну платформу.

У разі *сильної мобільності* основною проблемою, яку треба розв'язати, є перенесення сегмента виконання, оскільки він значною мірою залежить від платформи, на якій виконується завдання. На практиці перенести сегмент виконання, не вносячи до нього жодних змін, можна лише за умови, що машина-приймач має таку саму архітектуру і працює під керуванням тієї самої операційної системи.

Сегмент виконання містить закриті дані процесу, його поточний стек і лічильник програми. Поточний стек зазвичай містить тимчасові дані: значення локальних змінних, інформацію, що залежить від платформи, зокрема значення регістрів. Важливо відзначити, що якби вдалося позбавитися від даних, які залежать від платформи, то перенести сегмент на іншу машину і продовжити виконання там було б значно простіше.

Процедура перенесення коду обмежена декількома конкретними аспектами виконання програми. *Перенесення можливе лише у момент виклику*

чергової підпрограми, під якою розуміють функцію в *C*, метод у Java тощо. Виконавча система створює власну копію програмного стека, причому машинно-незалежну, яку називають *стеком перенесення (migration stack),* що оновлюється під час виклику підпрограми або повернення керування з підпрограми.

Під час виклику підпрограми виконавча система проводить **маршалінг даних,** поміщених у *стек перенесення* під час попереднього виклику (рис. 4.15). Ці дані є *значеннями локальних змінних* та *параметрів поточного виклику процедури*. Дані, які отримують після маршалінгу, теж розміщують *у стеку перенесення* разом з *ідентифікатором викликаної підпрограми* та *адресою у формі мітки переходу*, з якої має продовжуватися виконання після повернення з підпрограми.

Якщо перенесення коду відбувається в точці виклику підпрограми, то виконуюча система здійснює спочатку маршалінг усіх глобальних даних програми, які становлять сегмент виконання. Дані, властиві для цієї машини, і поточний стек ігноруються. Дані після маршалінгу і стек перенесення передаються на машину, яка їх очікує. На машину-приймач завантажується сегмент коду, що містить відповідні для її архітектури й операційної системи код. На машині-приймачі виконується демаршалінг отриманих даних сегмента виконання, та зі стека перенесення формується новий стек виконання. Після цього виконання може бути продовжене за рахунок входу в підпрограму, викликану на вихідному сайті.

Зрозуміло, що такий підхід можливий лише за умови, що компілятор генерує код для оновлення стека перенесення під час кожного входу в підпрограму або виходу з неї. Компілятор має також генерувати в коді *мітки*, які дозволяють реалізувати вихід з підпрограми у вигляді переходів (машинно-незалежних). Для функціонування процесу перенесення коду необхідна відповідна виконавча система. Є низка систем, які успішно використовують розглянуту технологію. Проблеми перенесення коду, зумовлені гетерогенністю розподілених систем, у багатьох випадках схожі з загальними проблемами перенесення, тому також подібні й методи їх вирішення.

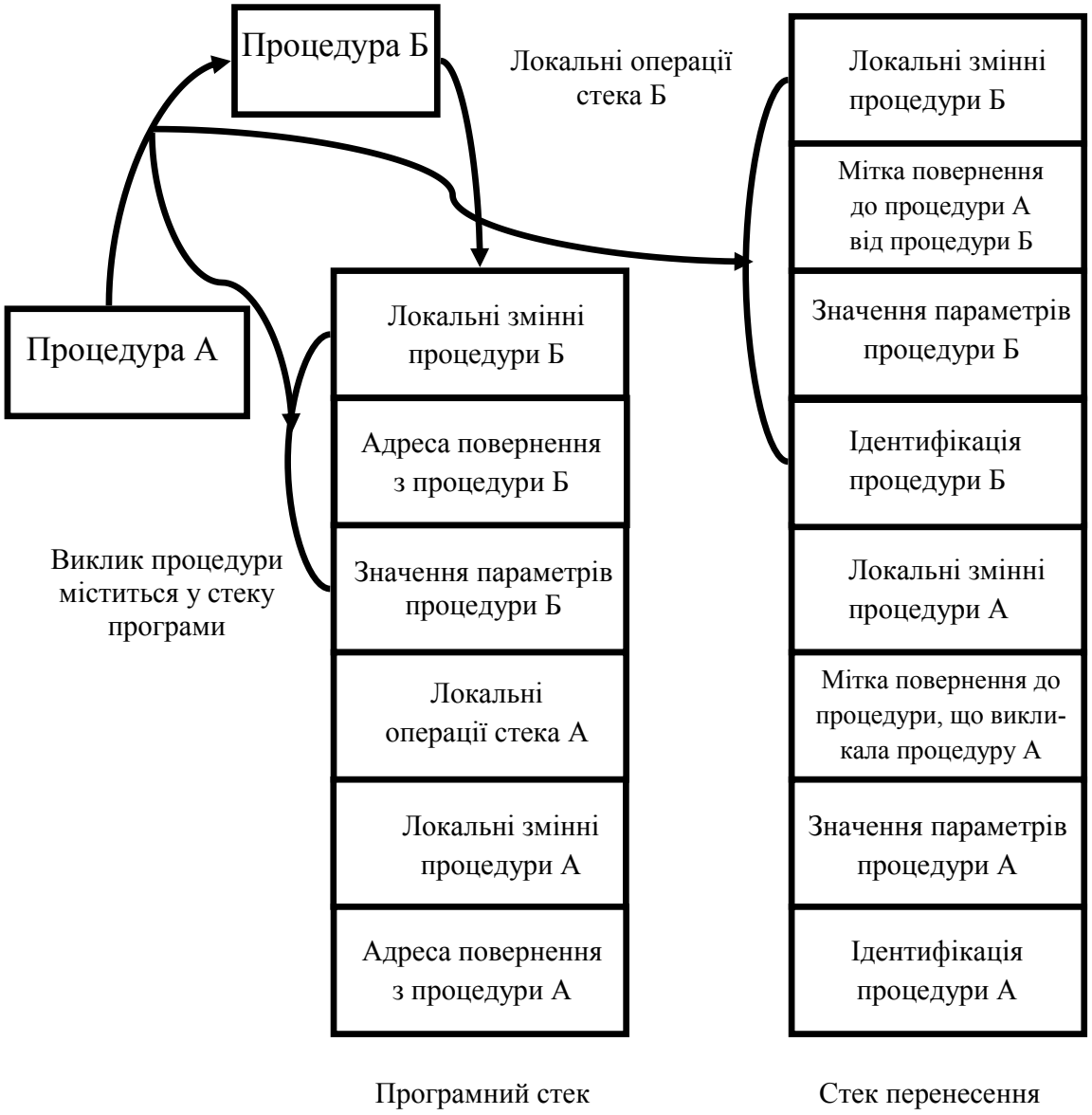


Рис. 4.15. Перенесення коду в гетерогенних системах

Приклад. У кінці 70-х років 20-го сторіччя було запропоновано просте рішення, що дозволило розв’язати низку проблем з перенесенням мови Pascal на різні машини, а саме генерація проміжного машинно-незалежного коду для абстрактної віртуальної машини, яка дозволила програмам мовою Pascal працювати на різних платформах. Цю просту ідею деякий час широко використовували, однак ніколи не вважали загальним вирішенням усіх проблем перенесення для інших мов, особливо *C*. На сьогодні проблему перенесення коду в гетерогенних розподілених системах почали розв’язувати засобами *мов сценаріїв* та мов, що мають *високу міру перенесення*, таких як *Java.*

Всі розглянуті підходи загалом, ґрунтуються на понятті «*віртуальна машина»,* яка інтерпретує або безпосередньо вихідні тексти програм (для мов сценаріїв), або проміжний код, що видається компілятором (для Java).

Єдиний суттєвий недолік перенесення, яке реалізують віртуальні машини, полягає в тому, що доводиться обмежуватися конкретною мовою програмування. З огляду на це важливо, щоб мови, призначені для перенесення, мали інтерфейс із наявними мовами.

4.9 Огляд перенесення коду в D'Agent

Щоб проілюструвати перенесення коду, розглянемо платформу проміжного рівня, що підтримує різні форми перенесення коду.

Agent TCL (D'Agent) – це система, побудована на основі концепції **агента,** яким у цій системі називають програму, яка в гетерогенній системі здатна переміщуватися з однієї машини на другу.

Програми можуть бути написані різними мовами за умови, щоб машина, на яку переноситься код, могла виконати його. На практиці це означає, що програми в D'Agent пишуться інтерпретованими мовами, зокрема командною мовою утиліт (Tool Command Language, TCL), Java або Scheme. Використання винятково інтерпретованих мов значно полегшує підтримку перенесення коду в гетерогенних системах.

Програма, або агент, виконується у процесі, запущеному *інтерпретатором мови*, якою ця програма написана. Мобільність підтримується трьома способами: за допомогою *ініційованої відправником слабкої мобільності, сильної мобільності з перенесенням процесів* і *сильної мобільності з клонуванням процесів*.

Слабка мобільність реалізується за рахунок команди «agent_submit», як параметр використовується *ідентифікатор машини*, на яку переноситься код і виконується *сценарій.*

Сценарій – *послідовність інструкцій*, яка переноситься на машину–одержувач разом з усіма описами процедур і копіями змінних, необхідними для його виконання на цій машині. На машині-одержувачі процес запускає відповідний інтерпретатор, який і виконує сценарій. D'Agent забезпечує ініці-

йовану відправником слабку мобільність, коли перенесений код виконується в окремому процесі.

Сильна мобільність, яку ініціює відправник у формі як *міграції,* так і *клонування процесів* виконується за таким алгоритмом. Для перенесення агента, який працює, відправник викликає команду «*agent_jump»* із зазначенням цільової машини, на яку він має бути перенесений. Під час виклику команди «*agent_jump»* виконання агента на вихідній машині припиняється, і його сегмент ресурсів, сегмент коду і сегмент виконання піддаються маршалінгу, пакуючись у *повідомлення*, яке потім пересилається на цільову машину. Після доставки повідомлення запускається новий процес, у якому активується відповідний інтерпретатор, що виконує *демаршалінг повідомлення*, яке надійшло, потім продовжує виконання з інструкції, наступної за останнім викликом «*agent_ jump»*. Процес, у якому агент виконувався на вихідній машині, припиняє свою роботу.

Алгоритм підтримки *клонування процесів* за рахунок команди «*agent_fork*» працює майже так само, як і «*agent_jump»*, за винятком того, що процес, який запустив агента на вихідній машині, продовжує роботу за інструкцією, наступною за викликом «*agent_fork»*. Як і в операції «*fork»* у UNIX-системах, команда «*agent_fork»* повертає значення, за яким процес, який її викликає, може визначити, що перед ним: клонована версія («дочірня») чи оригінальний процес («батьківський»).

Розглянемо деякі деталі внутрішньої реалізації на прикладі агентів, яких написано мовою TCL. Система D’Agent складається *з п’яти рівнів* (рис. 4.16). Найнижчий рівень аналогічний сокетам Берклі тим, що реалізує єдиний інтерфейс механізмів взаємодії базової мережі. У D’Agent передбачається, що базова мережа надає механізми для роботи з повідомленнями TCP і електронної пошти.

Наступним рівнем є сервер, що працює на кожній машині, на якій виконується D’Agent. Сервер відповідає за керування агентами, авторизацію і керування зв’язком між агентами; призначає кожному агентові *локальний унікальний ідентифікатор*. Якщо використовувати мережну адресу сервера, то кожен з агентів може бути позначений *парою (адреса, локальний іденти-*

фікатор). Таке ім'я низького рівня використовують *для встановлення зв'язку* між двома агентами.

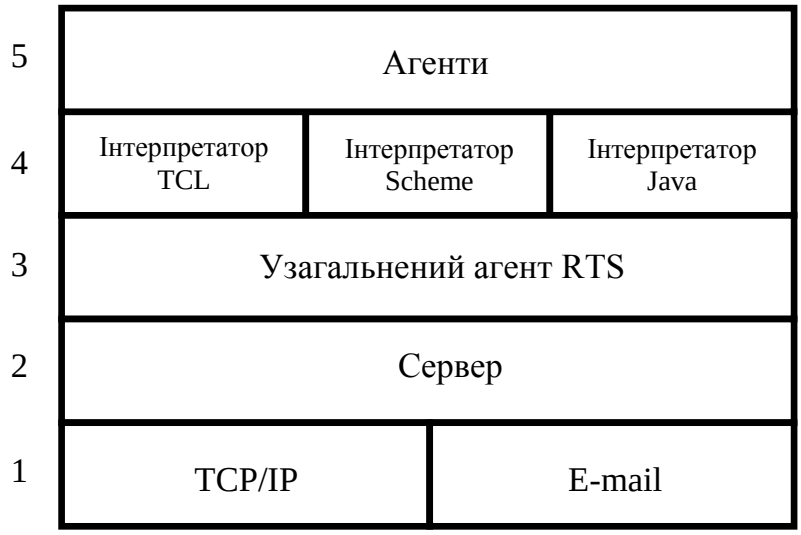


Рис. 4.16. Склад системи D'Agent

Третій рівень має незалежне від мови ядро, яке підтримує основні моделі агентів; містить *реалізацію запуску і закінчення роботи агента, реалізації різних операцій перенесення й засобу для зв'язку між агентами*. На відміну від сервера, ядро не відповідає за керування набором агентів, розміщених на одній машині.

Четвертий рівень має по одному інтерпретатору на кожну підтримувану в D'Agent мову. Кожен інтерпретатор містить *компонент інтерпретації мови, модуль безпеки, інтерфейс із рівнем ядра й окремий модуль для перехоплення стану працюючого агента*, необхідного для підтримки сильної мобільності.

Найвищий рівень містить агентів, написаних однією з підтримуваних системою мов, кожен агент D'Agent виконується в окремому процесі.

Приклад. Коли агент переноситься на машину A, сервер розгалужує процес виконання відповідного інтерпретатора, створюючи гілку для виконання цього агента. Новий процес підхоплює стан агента, що мігрував, і продовжує його виконання з тієї точки, на якій він був призупинений. Сервер відстежує локальні канали створеного процесу, за допомогою яких процес отримує призначені для нього повідомлення.

Найскладніша частина реалізації D'Agent – це *отримання стану працюючого агента і передача його на другу машину*, для чого інтерпретатору необхідна таблиця, в якій зберігаються глобальні змінні. Так, обробник подій може

повідомляти інтерпретатора, яку процедуру слід викликати в разі надходження повідомлення від якогось агента. Пари (подія, обробник) зберігають у таблиці інтерпретатора. У другій таблиці містяться глобальні системні змінні, в яких зберігають коди помилок, рядки з повідомленнями про помилки, коди результатів, рядки повідомлень, що виводяться разом з результатами, тощо. Є також окрема таблиця, в якій зберігають усі визначені користувачем глобальні змінні програми, визначення процедур, пов'язаних з агентами, що потребують перенесення разом з агентами для вибору інтерпретатора на цільовій машині.

Розглянемо аспект, безпосередньо пов'язаний з перенесенням агентів, – *наявність двох стеків*, у яких зберігається поточний стан виконання агента. В основу кожного агента покладено набір команд TCL, вбудованих у такі конструкції, як цикли, інструкції множинного вибору, команди, згруповані у процедури. Інтерпретація такого набору команд агентом виконується послідовно, команда за командою.

Спочатку розглянемо, що відбувається під час виконання базової команди TCL, тобто команди, яка викликається не з призначеної для користувача процедури. Інтерпретатор аналізує команду і будує запис, який поміщається у **стек команд** *(command stack)* та містить усі необхідні для виконання команди поля: *значення параметрів, вказівник на процедуру реалізації команди та інші*. Коли запис поміщено у стек, то після цього він може бути використаним компонентом, який відповідає за виконання команди, тобто стек команд є місцем зберігання поточного стану виконання агента.

Команда TCL також підтримує процедури, які визначає користувач. Середовище виконання D'Agent відстежує *стеків команд* та *записів активізації*, які також називають фреймами виклику. **Фрейм виклику** в D'Agent містить таблицю змінних, локальних для процедури, імена і значення параметрів, з якими ця процедура була викликана. Фрейм виклику створюється лише в результаті виклику процедури і стосується команди виклику процедури, поміщеної у стек команд. Фрейм виклику містить посилання на пов'язану з ним команду.

Приклад. Розглянемо, що відбувається, коли агент викликає команду «*agent_jump»*, за допомогою якої він переноситься на іншу машину. В момент виклику команди стан

агента піддається маршалінгу і перетворюється в послідовність байтів, тобто, всі чотири таблиці та два стеки укладаються в масив байтів і пересилаються. Сервер D'Agent на цільовій машині створює новий процес, запускаючи інтерпретатор TCL. Процес обробляє отримані дані, виконує їх демаршалинг і в результаті набуває стану агента, у якому він перебував перед викликом команди «*agent_jump»*. Виконання програми-агента продовжується за рахунок простого зняття чергової команди з вершини стека команд.

4.10 Програмні агенти

З позицій взаємодії програмних компонентів у мережі розглянемо процеси, керівні потоки виконання всередині процесів, узагальнену організацію клієнтів і серверів, а також перенесення програм і процесів. Незалежні процеси об'єднаються в те, що називають **програмними агентами** – автономними одиницями, здатними виконувати завдання в кооперації з іншими, можливо, віддаленими агентами. Агенти відіграють у розподілених системах важливу роль.

4.10.1 Програмні агенти в розподілених системах

Програмний агент *(software agent)* – це автономний процес, здатний реагувати на середовище виконання і змінювати його, можливо, в кооперації з користувачами або з іншими агентами.

Властивість, яка відрізняє агента від процесу, – це здатність функціонувати автономно і, зокрема, проявляти у разі потреби ініціативу.

Визначення програмного агента є достатньо неточним, у результаті чого багато типів процесів легко можуть сприйматися як агенти. Оскільки важко достатньо точно встановити поняття «програмні агенти», тому варто з'ясувати різні типи програмних агентів, класифікація яких зумовлена їх властивостями.

Окрім *автономності,* важливою якістю агентів є *можливість кооперуватися з іншими агентами*. Поєднання автономності й кооперації зумовлює появу *кооперативних агентів.*

Кооперативний агент *(collaborative agent)* – це агент, яке становить частину мультиагентної системи, тобто системи, в якій агенти, кооперуючись, виконують деякі загальні завдання.

Приклад. Типове програмне забезпечення, що використовує кооперативних агентів, – це електронна конференція. Кожного доповідача представляє агент, що має доступ до матеріалів, які користувач хоче надати до загального розгляду. З урахуванням усіх індивідуальних обмежень щодо часу, місця розташування, переміщення та інших спільна робота окремих агентів дозволяє організувати конференцію.

У деяких випадках виділяють серед інших типів агентів мобільні агенти. **Мобільний агент** *(mobile agent)* – це агент, який здатний переміщуватися з машини на машину.

З огляду на положення перенесення коду мобільні агенти часто вимагають підтримки сильної мобільності, хоча це і не є абсолютно необхідним. Вимога сильної мобільності випливає з того, що агенти автономні та активно взаємодіють зі своїм середовищем, тому перенесення агента на іншу машину без урахування його стану буде значно утруднено. Проте, як було показано на прикладі системи D'Agent, поєднання агентів і слабкої мобільності також цілком можливе. Відзначимо, що **мобільність** – це загальна властивість агентів, наявність якої не є ознакою для віднесення їх до певного класу, тому має сенс говорити про існування *мобільних кооперативних агентів*, які можуть бути наявними в усіх класах.

Здатність до кооперації з іншими агентами або переміщення між машинами – це системні властивості агентів, що не визначають призначення агента. В основу класифікації агентів як ознаку покладено саме їх функціональність.

Класи, які традиційно розглядають, – це **інтерфейсні** та **інформаційні агенти**, а також можливо виділити інші класи за функціональними ознаками.

Інтерфейсний агент (*interface agent*) – це агент, що допомагає кінцевому користувачу працювати з одним або декількома програмними засобами.

Серед властивостей, які традиційно є в інтерфейсного агента, можна виокремити здатність до навчання. Найчастіше властивості взаємодіють з користувачами, забезпечуючи їм підтримку в процесі подання інформації у зручному для кінцевого користувача вигляді.

Приклад. У контексті розподілених систем зразком інтерфейсного агента може бути агент, що відстежує взаємодії між агентами і користувачами. Так створюються спеціальні інтерфейсні агенти для взаємодії продавців з покупцями. Правильно зрозумівши, що хоче

побачити покупець або що може запропонувати його власник, інтерфейсний агент може допомогти визначити потрібну групу товарів.

Інформаційний агент *(information agent)* – агент, що керує інформацією з безлічі різних джерел та подібний до інтерфейсного агента.

Керування інформацією охоплює впорядкування, фільтрацію, порівняння тощо. Необхідність використання інформаційних агентів у розподілених системах визначається тим, що вони можуть працювати з інформацією, яка збирається з різних територіально розподілених джерел. Стаціонарні інформаційні агенти зазвичай працюють з вхідними потоками інформації, яка надходить до певного вузла системи.

Приклад. Поштовий агент може застосовуватися для фільтрації в поштовій скриньці її власника небажаної кореспонденції або для автоматичного перенапрямлення вхідної пошти до поштових скриньок, які відповідають її темі.

На противагу стаціонарним агентам, мобільні інформаційні агенти зазвичай вільно подорожують мережею, збираючи на вимогу їх власника необхідну інформацію.

У загальному випадку агенти можна охарактеризувати низкою властивостей, наведених у табл. 4.2. Подальший поділ агентів можна виконати, розглядаючи, як вони працюють, використовуючи методи штучного інтелекту.

Таблиця 4.2. Деякі важливі властивості агентів

Властивість	**Властивості, спільні для агентів**	**Опис**
Автономність	є	Здатність працювати незалежно від інших
Реактивність	є	Здатність своєчасно реагувати на зміни у своєму оточенні
Функціональність	є	Здатність ініціювати дії, що впливають на оточення агентів
Комунікативність	є	Здатність обмінюватися інформацією з користувачами та іншими агентами
Тривалість	Немає	Порівняно довгий час життя
Мобільність	Немає	Здатність переміщуватися з місця на місце
Адаптивність	Немає	Здатність до навчання

4.10.2 Технологія агентів

Розгляд сутності програмних агентів є непотрібним, якщо без них не утворювали реальних систем агентів. Підвищення ефективності функціонування розподілених систем можливе у разі виділення постійно використовуваних компонентів агентів у таких системах та інтегрування їх до програмного забезпечення проміжного рівня. Організація FIPA (Foundation for Intelligent Physical Agents) розробила **узагальнену модель програмних агентів**, згідно з якою агенти реєструються і працюють під керуванням **платформи агентів,** як показано на рис. 4.17. Платформа агентів надає основні служби, необхідні будь-якій мультиагентній системі, зокрема механізми створення і знищення агентів, механізми розпізнавання агентів і механізми взаємодії між агентами.

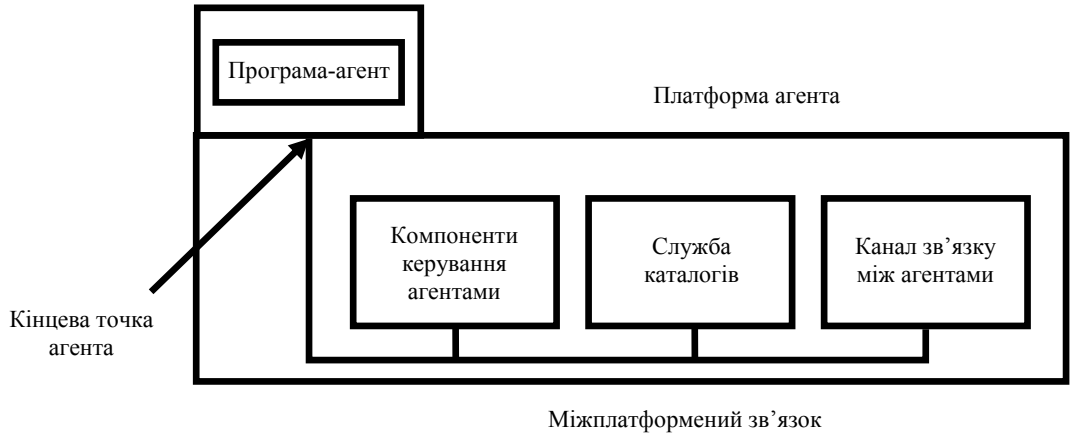


Рис. 4.17. Узагальнена модель платформи агента

Компонент керування агентами відстежує агентів на відповідній платформі, надає *механізми створення і знищення агентів*, а також механізм перегляду поточної кінцевої точки щодо наявності конкретного агента, тобто надає *службу найменування*, за допомогою якої *глобально унікальний ідентифікатор* відображується *на локальній кінцевій точці взаємодії.*

До складу узагальненої моделі платформи агента входить окрема *локальна служба каталогів*, за допомогою якої агенти можуть дізнатись, які ще агенти є на цій платформі. В основі служби каталогів у моделі FIPA лежить використання *атрибутів*, тобто агент надає описи своїх служб у вигляді імен атрибутів разом

з їх значеннями для певного агента. До служби каталогів можуть мати доступ віддалені агенти, тобто агенти, що перебувають на інших платформах агентів.

Важливий компонент платформи агента – *канал зв'язку між агентами* ACC (Agent Communication Channel). У більшості моделей мультиагентних систем агенти взаємодіють один з одним, пересилаючи повідомлення. Модель FIPA – не виняток, вона використовує ACC для взаємодії між різними платформами агентів, надає надійний і напрямлений зв'язок «точка–точка». Канал ACC може бути реалізований у вигляді сервера, що відстежує деякий порт, призначений для вхідних повідомлень. Повідомлення перенапрямляються іншим серверам або агентам, які є частиною платформи агентів. Для забезпечення міжплатформної взаємодії зв'язок між ACC має відповідати Internet–протоколу IIOP (Internet INTER-ORB Protocol). В архітектурі D'Agent роль ACC виконує сервер.

4.10.3 Мови взаємодії агентів

У платформі агентів є своя специфіка. Відмінність агентного підходу від інших підходів до побудови розподілених систем стає зрозумілою під час розгляду характеру інформації, якою обмінюються агенти. Зв'язок між агентами відбувається за допомогою *комунікаційного протоколу прикладного рівня*, відомого під назвою *мови взаємодії агентів ACL (Agent Communication Language)*. В ACL властиво чітке розділення між метою повідомлення і його змістом. Повідомлення може мати лише обмежений набір цілей.

Приклад. Метою повідомлення може бути запит на надання отримувачем певної служби, відповідь на раніше надіслане повідомлення із запитом, а також повідомлення сторони, яка його приймає, про подію, що сталася, або про певну пропозицію в ході узгодження.

В ACL агент-відправник і агент-одержувач однаково розуміють мету повідомлення, від якої зазвичай залежить реакція одержувача.

Приклад. Так, у разі запиту пропозиції через повідомлення, що має в заголовку мету *CFP*, одержувач повинен надіслати пропозицію за допомогою повідомлення з метою *PROPOSE*. У цьому сенсі ACL відіграє роль високорівневого комунікаційного протоколу для набору агентів.

Як і більшість комунікаційних протоколів, повідомлення ACL складаються *із заголовка й реального вмісту* (рис. 4.18). **Заголовок** містить *поле мети повідомлення, поля відправника й одержувача*. Крім того, як і в багатьох інших комунікаційних протоколах, уміст листа є відокремленим і незалежним від іншої його частини, тобто його визначать агенти, які встановили зв'язок. Повідомлення ACL не задає формату або мови змісту.

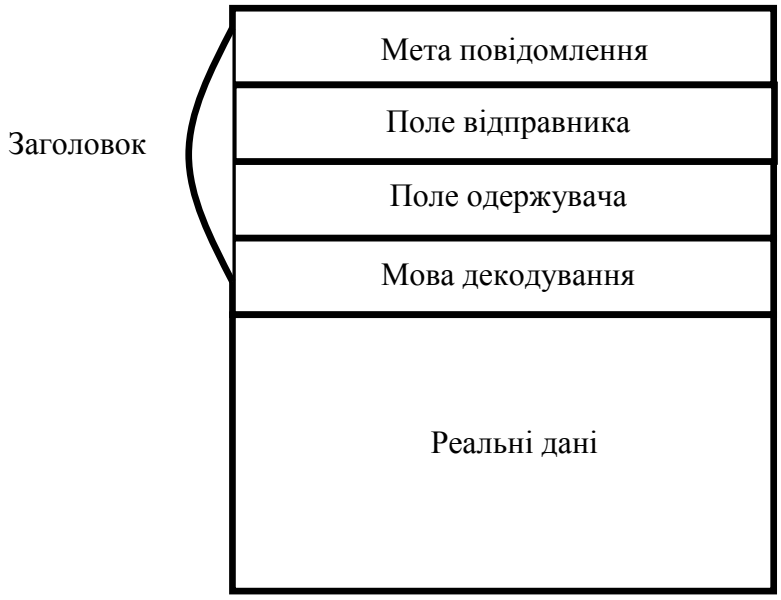


Рис. 4.18. Структура повідомлення ACL

У разі встановлення зв'язку між агентами необхідно, щоб агентові, який приймає, було надано всю необхідну інформацію для правильної інтерпретації вмісту, для чого заголовок повідомлення ACL має також визначати *мову* або *схему декодування* вмісту. Цей підхід добре працює доти, доки відправник і одержувач однаково інтерпретують дані, якими є символи повідомлення. Якщо єдиного розуміння немає, то потрібні додаткові поля, що ідентифікують *стандартне відображення символів у їх зміст,* яке зазвичай називають *онтологією (ontology).*

4.11 Висновки

1. Процеси відіграють фундаментальну роль у розподілених системах, оскільки вони формують базис для зв'язку між різними машинами.

2. Важливою є внутрішня організація процесів, зокрема чи здатні вони підтримувати декілька керівних потоків виконання, які в розподілених системах використовуються для продовження роботи з процесором під час блокуючих операцій введення-виведення. Ефективною є побудова серверів, у яких декілька потоків виконання працюють одночасно, причому деякі з них можуть блокуватися в очікуванні виконання дискових операцій введення–виведення або операцій мережної взаємодії.

3. Організацію розподілених програмних комплексів виконують у вигляді клієнтів і серверів. Клієнтський процес зазвичай реалізує призначений для користувача інтерфейс, який може варіюватися від простого виведення інформації до розширених інтерфейсів, здатних підтримувати документи складної структури. Клієнтське програмне забезпечення здатне підтримувати прозорість розподілу, приховуючи деталі, що стосуються зв’язку із серверами, поточного місця розташування серверів і реплікації серверів, а також частково приховати збої та процеси відновлення після збоїв.

4. У процесі побудови серверів застосовують різні архітектурні моделі. Сервери можуть бути інтерактивними або паралельними, реалізовувати одну або декілька служб, зберігати інформацію про стан або не зберігати. Архітектурні особливості впливають на адресацію служб і механізмів переривання серверів після надходження запиту на обслуговування й у ході його виконання.

5. Сервери об’єктів виокремлено в окремий клас. Сервер об’єктів – це процес, який містить об’єкти, розміщені в його власному адресному просторі, і готовий приймати напрямлені до них звертання. В окрему категорію сервери об’єктів виділяють завдяки різноманітності способів звертання до об’єктів. Сервер може запустити окремий потік виконання для кожного запиту до об’єкта, але й може надати кожному об’єкту власний потік виконання або залишити єдиний потік виконання для всіх своїх об’єктів. За допомогою адаптера об’єктів у різних серверах може бути реалізована різна політика звертання до об’єкта. На сервері може бути декілька адаптерів об’єктів.

6. Важливою для розподілених систем функціональною властивістю є перенесення коду з машини на машину, для застосування якого є дві вагомі

причини – підвищення продуктивності й мобільність. Інколи є потреба зменшити рівень взаємодії за рахунок перенесення обчислень із сервера на клієнта. Гнучкість зростає, коли клієнт має можливість динамічно завантажувати програмне забезпечення, необхідне для роботи з конкретним сервером. Завантажене програмне забезпечення може бути вже налаштованим на взаємодію із цим сервером, позбавляючи клієнта потреби повторно встановлювати його до початку роботи.

7. Під час перенесення коду виникає проблема, пов’язана з використанням локальних ресурсів, тому що ці ресурси також необхідно переносити на інші машини, організовувати нові прив’язки коду до локальних ресурсів цільової машини або застосовувати глобальні посилання. Друга проблема полягає в тому, що під час перенесення коду слід ураховувати гетерогенність системи. Практика показує, що найкраще слід будувати систему в гетерогенному середовищі, використовуючи віртуальні машини, які ефективно приховують гетерогенність за допомогою коду, що інтерпретується.

8. Програмні агенти – спеціальний вид процесів, що працюють як автономні модулі, здатні кооперуватися з іншими агентами. З точки зору розподілених систем, відмінність агентів від звичайних програмних засобів полягає у тому, що агенти взаємодіють один з одним за допомогою комунікаційного протоколу прикладного рівня, який називають мовою взаємодії агентів (ACL) та у якому передбачено чітке відокремлення мети повідомлення і його змісту.

4.12 *Запитання для самоконтролю*

1. Що розуміють під поняттям «процес»?
2. Що таке потоки виконання?
3. З яких елементів складається потік виконання?
4. Побудуйте схему потоку виконання та поясніть її.
5. Назвіть стани, в яких можуть перебувати процеси та потоки виконання.
6. Назвіть способи організації побудови серверів. У чому полягають відмінності між ними?

7. Охарактеризуйте організацію багатопотокового сервера.
8. Назвіть завдання клієнтського програмного забезпечення.
9. Як реалізується прозорість реплікації на стороні клієнта?
10. Що розуміють під поняттям «сервер»?
11. Які розрізняють способи організації серверів?
12. Що таке сервер об’єктів?
13. Які є підходи до обробки об’єктів?
14. Обґрунтуйте необхідність перенесення коду.
15. Сформулюйте основні підходи до перенесення коду.
16. У чому полягає процедура перенесення локальних ресурсів?
17. Як реалізується перенесення коду в гетерогенних системах?
18. Як реалізується перенесення коду в D’Agent?
19. Що розуміють під поняттям «агент»?
20. Які розрізняють види агентів?
21. У чому полягає технологія агентів?

5. ІМЕНУВАННЯ

Імена відіграють важливу роль у всіх комп'ютерних системах, оскільки необхідні для спільного використання ресурсів, визначення унікальних сутностей, посилань на місця розташування тощо. Важлива особливість *іменування* полягає в тому, що *ім'я може ідентифікувати сутність*, надаючи доступ до неї.

Ідентифікація імені *являє собою процес доступу до іменованої сутності.* Для ідентифікації імен необхідно реалізувати іменування. Відмінність між іменуванням у розподілених і нерозподілених системах полягає у *способі реалізації систем іменування.*

У розподілених системах реалізація системи іменування часто розподілена між декількома машинами. Спосіб цього розподілу відіграє істотну роль для ефективності й масштабованості системи імен. Розглянемо три різні, але однаково важливі способи використання імен у розподілених системах.

1. До розгляду деяких загальних питань, пов'язаних з іменуванням, проаналізуємо організацію й реалізацію *імен* «*людей*», типові приклади яких містять імена файлової системи й World Wide Web (WWW). Побудова глобальної масштабованої системи іменування ґрунтується на типах імен.

2. Імена використовують для *локалізації* мобільних сутностей. Системи іменування на основі імен «людей» не підходять для підтримки значної кількості мобільних сутностей, які можуть бути розкидані глобальною мережею. Необхідна альтернативна організація, аналогічна тій, що використовується в мобільній телефонії, у якій *імена (ідентифікатори) не залежать від місця розташування*.

3. Особливості *організації імен*. Імена, на які вже не посилається жоден об'єкт і яких можна у майбутньому не локалізувати, щоб одержати до них доступ, мають автоматично видалятися. Цей процес називають *збиранням сміття*. Під час переходу до великих розподілених систем автоматичне збирання об'єктів, на які немає посилань, стає особливо важливим.

5.1 Іменовані сутності

5.1.1 Імена, ідентифікатори й адреси

Ім'я *в розподіленій системі являє собою послідовність бітів (символів), які використовуються для посилання на сутність.*

Сутністю в розподіленій системі є майже все. Типовими прикладами сутностей є такі ресурси, як хости, принтери, диски, файли. Над сутностями можна виконувати *операції*, але й сама сутність може *виконувати операції* над іншими сутностями.

Приклад. Принтер надає інтерфейс, що підтримує операції друкування документа і запиту стану друкування, працює з мережею. Разом з тим, така сутність, як мережне з'єднання, може здійснювати операції з передачі й прийому даних, установки параметрів якості обслуговування, запиту стану тощо.

Щоб працювати із сутністю, необхідно мати до неї доступ, для якого використовується *точка доступу (access point)*.

Точка доступу – це ще один спеціальний вид сутності в розподілених системах. Ім'я точки доступу називають **адресою (address). Адресу точки доступу сутності** часто називають просто *адресою сутності*. Сутність може мати більш ніж одну точку доступу.

Приклад. Для порівняння, телефон можна вважати точкою доступу до людини, якщо телефонний номер розглядати як адресу, однак нині більшість людей має кілька телефонних номерів, кожний з яких відповідає одній точці доступу, в якій цих людей можна застати. У розподілених системах типовим прикладом точки доступу є хост, на якому запущений якийсь сервер, адреса якого формується сполученням, наприклад, IP-адреси й номера порту (тобто адресою сервера транспортного рівня).

Точка доступу сутності може із часом змінюватися.

Приклад. Якщо мобільний комп'ютер перемістити в іншу безпроводову мережу, то йому буде присвоєно іншу IP-адресу.

Адреса – це спеціальний тип імені, який вказує на точку доступу до сутності. Оскільки точка доступу тісно пов'язана із сутністю, зручно використовувати адресу як постійне ім'я відповідної сутності, однак робити це можна не завжди.

У трактуванні адрес як імен особливого типу є багато переваг. Так, у разі реорганізації розподіленої системи сервер може опинитися на іншому хості, тобто сутність змінить точку доступу, а поточна точка доступу буде перенацілена на іншу сутність.

Якщо для посилання на сутність використовувати адресу, то у разі зміни точки входу або призначення їй іншої сутності одержимо помилкове посилання.

Приклад. Уявімо, що FTP-служба організації відома лише за адресою хоста, на якому запущений FTP-сервер. Як тільки сервер переведено на інший хост, FTP–служба буде зовсім недоступною доти, доки нова адреса не стане відомою всім користувачам цієї служби. За таких умов значно краще зробити так, щоб FTP–служба мала власне ім'я, ніяк не пов'язане з адресою відповідного FTP–сервера. Аналогічно, якщо сутність має більше однієї точки входу, то незрозуміло, за якою адресою посилатися, наприклад для web-служби краще використовувати одне ім'я, не пов'язане з адресами web-серверів.

Імена сутності, не пов'язані з її адресами, називають *локально незалежними (local independent),* їх використання є значно зручнішим.

Крім адрес, розрізняють й інші типи імен, зокрема використовувані для *однозначної ідентифікації сутності.* **Правильний ідентифікатор** (*true identifier*) – це ім'я, що має такі властивості:

- ідентифікатор посилається не більше, ніж на одну сутність;
- на кожну сутність посилається не більше одного ідентифікатора;
- ідентифікатор завжди посилається на ту саму сутність, тобто не може бути використаним повторно.

Використання ідентифікаторів істотно спрощує створення однозначних посилань на сутність.

Приклад. Розглянемо два процеси, кожний з яких посилається на певні сутності за допомогою ідентифікаторів. Для того щоб зрозуміти, що обидва процеси посилаються на ту саму сутність, достатньо порівняти чи є еквівалентними два ідентифікатори. У разі використання звичайних (не «ідентифікаторних») імен такого тесту замало.

Аналогічно, якщо одна й та ж адреса може бути призначена іншій сутності, то її не можна використовувати як ідентифікатор.

Приклад. Розглянемо застосування телефонних номерів, які порівняно постійні, тому що номер зазвичай відповідає одній людині або організації. Однак використову-

вати телефонний номер як ідентифікатор не можна, тому що він із часом може змінитися.

Адреси й ідентифікатори – два дуже важливих типи імен, кожний з яких призначений для своїх цілей. У багатьох комп'ютерних системах адреси й ідентифікатори зберігаються лише у формі, зручній для використання машиною, тобто у формі рядків бітів.

Ще один суттєвий тип імен – *імена, пристосовані для сприйняття людиною (human-friendly names)*. На противагу адресам й ідентифікаторам, зручне для сприйняття ім'я зазвичай подається у вигляді рядка символів. Ці імена набувають безліч різних форм.

Приклад. Файли в UNIX-системах мають імена, що складаються з рядків, довжина яких може досягати до 255 символів і визначає довжину імені користувача за власним бажанням. Аналогічно, імена DNS подаються як порівняно простий рядок, символи в якому не залежать від регістру.

5.1.2 Простір імен

Імена в розподілених системах організуються в деяку сутність **– простір імен** (*name space*), який імен може бути поданим як напрямлений граф із двома типами вузлів. **Листовий вузол** *(leaf node)*, або лист, який представлений іменованою сутністю, не має вихідних із нього ребер, зазвичай містить інформацію про сутність, яка представляється (наприклад, її адреса) та до якої мають доступ клієнти. Крім того, можна зберегти в листовому вузлі *стан цієї сутності*, наприклад у разі файлової системи, листовий вузол містить файл, який він представляє.

На відміну від листового, **напрямний вузол (directory node)** має декілька вихідних ребер, кожне із яких іменовано, як показано на рис. 5.1. Кожний вузол у графі іменування розглядається як ще одна сутність розподіленої системи, тобто має окремий ідентифікатор. Напрямий вузол зберігає *таблицю*, у якій усі вихідні ребра подані у вигляді пар (мітка ребра, ідентифікатор вузла) та яку називають *напрямною таблицею (directory table).*

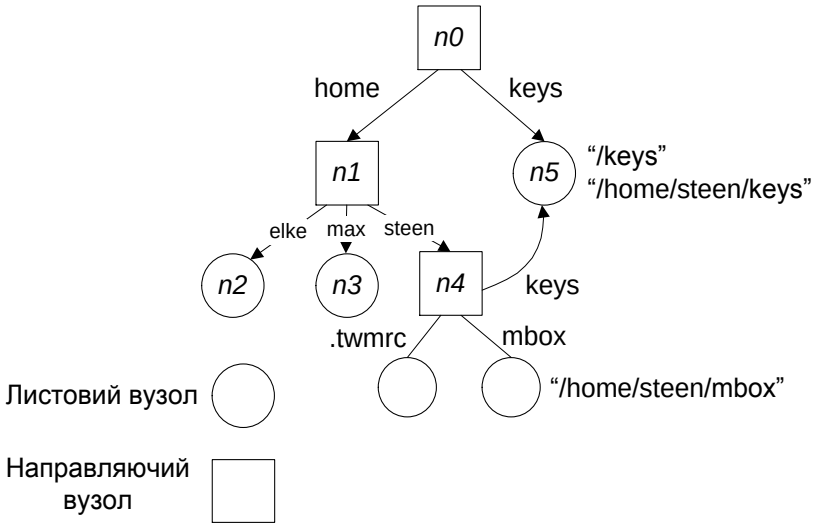


Рис. 5.1. Узагальнений граф іменування з одним кореневим вузлом

Граф іменування, показаний на рис. 5.1, має один вузол, саме той, у якого є тільки вихідні ребра, а вхідних немає. Такий вузол називають **кореневим вузлом** *(root node),* або *коренем графа іменування*. Незважаючи на те, що граф іменування може мати кілька кореневих вузлів, багато систем іменування мають лише один корінь. Будь-який *шлях* у графі іменування має бути поданий як *послідовність міток*, що відповідають ребрам графа.

Приклад. *N:< label-1, label-2 label-n>*, де *N* відповідає першому з вузлів шляху.

Послідовність, аналогічну поданій у прикладі, називають **іменем шляху** *(path name)*. Якщо *перший вузол шляху – корінь графа іменування*, то це ім'я називають **абсолютним іменем шляху** *(absolute path name),* в іншому разі – **відносним іменем шляху** *(relative path name).*

Важливо розуміти, що імена завжди організуються **у простір імен,** унаслідок чого ім'я завжди визначається тільки щодо напрямного вузла, тому термін «абсолютне ім'я» – це не в повному сенсі «абсолютне ім'я», так само й відмінність між глобальними й локальними іменами інколи може ввести в оману.

Глобальним *(global name)* називають ім'я, що позначає ту саму сутність, незалежно від того, де в системі воно використовується, тобто глобальне ім'я завжди відповідає тільки одному напрямному вузлу.

На відміну від глобального, **локальне ім'я** *(local name)* – це ім'я, інтерпретація якого залежить від того, де воно використовується, тобто локальне

ім’я – це лише відносне ім’я, і має бути відомим напрямний вузол, якого воно стосується.

5.1.3 Реалізація просторів імен

Простір імен формує основу *служби іменування*, яка дозволяє користувачам і процесам додавати, видаляти й знаходити імена. Служба іменування реалізується за допомогою *серверів імен*. Якщо розподілену систему зменшено до розмірів локальної мережі, то вона спроможна реалізувати службу іменування за допомогою лише одного сервера імен. Однак у великих розподілених системах із багатьма сутностями, розміщеними в різних географічних зонах, необхідно рознести реалізацію простору імен по декільком серверам імен.

Простір імен великомасштабних або глобальних розподілених систем зазвичай організовано ієрархічно. Передбачається, що простір імен має *тільки один кореневий вузол*. Для ефективної реалізації цього простору імен його поділяють на такі *логічні рівні*: *глобальний рівень*, *адміністративний рівень*, *рівень керування*.

Реалізація ідентифікації імен. Розподіленість простору імен по безлічі серверам імен ускладнює реалізацію *ідентифікації імен*. Щоб пояснити реалізацію ідентифікації у великих службах імен, уявімо, що сервери не реплікуються і не кешуються на стороні клієнта. Кожний клієнт має доступ до *локальної процедури ідентифікації імен (name resolver)*, яка і відповідає за цей процес.

Приклад. Припустимо (рис. 5.2), що ідентифікується (абсолютний) шлях *root:<nl, vu, cs, ftp, pub, globe, index.txt>*. Якщо використовувати форму запису URL, то цей шлях відповідає імені *ftp://ftp.cs.vu.nl/pub/globe/index.txt*.

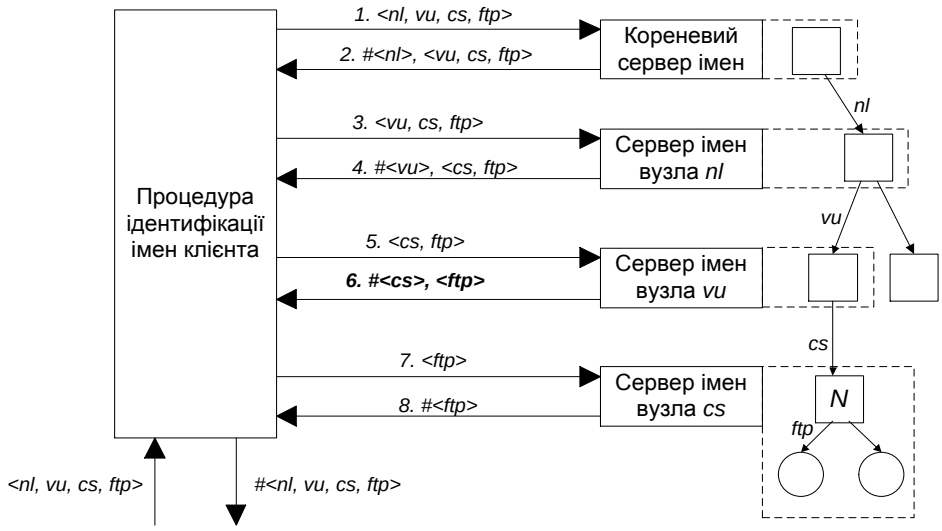


Рис. 5.2. Принцип ітеративної ідентифікації імен

Розрізняють два способи реалізації ідентифікації імен. У разі *ітеративної ідентифікації імен (iterative name resolution)* процедура передає *повне ім'я* кореневому серверу імен, тобто передбачено, що адреса кореневого сервера, з яким контактує процедура ідентифікації імен, є загальновідомою. Кореневий сервер ідентифікує ту частину шляху, яка зможе виконати запит і повернути результат клієнтові.

Приклад. У попередньому прикладі кореневий сервер може ідентифікувати лише мітку *nl*, для якої він і поверне адресу асоційованого з нею сервера імен. Після цього клієнт передасть цьому серверу імен частину шляху, що залишилася (тобто ім'я *nl:<vu, cs, ftp, pub, globe, index.txt>*). Сервер зможе ідентифікувати лише мітку *vu* і поверне адресу асоційованого із цією міткою сервера імен разом із частиною шляху, яка залишилася, тобто *vu:<cs, ftp, pub, globe, index.txt>*. Процедура ідентифікації імен клієнта зв'яжеться з наступним сервером імен, який зможе ідентифікувати мітку *cs*, а також з FTP-сервером, і поверне адресу FTP-сервера разом зі шляхом *ftp:<pub, globe, index.txt>*. Після цього клієнт зв'яжеться з FTP-сервером, вимагаючи від нього ідентифікувати залишок вихідного шляху, FTP-сервер ідентифікує послідовно мітки *pub*, *globe* та *index.txt* і передасть запитаний файл (у цьому разі за протоколом FTP). Такий процес ітеративної ідентифікації ілюструє рис. 5.2 (запис *#<cs>* використовується для зазначення адреси сервера, відповідального за обробку мітки *<cs>*).

На практиці останній крок, а саме зв'язок із FTP-сервером і запит на передачу файлу зі шляхом *ftp:<pub, globe, index.txt>*, виконується клієнтським процесом окремо від усіх інших процесів, тобто клієнт звичайним способом обробляє тільки шлях *root:<nl, vu, cs, ftp>*, з якого він вилучає адресу, за якою перебуває FTP-сервер (рис. 5.2)

Альтернативою ітеративної ідентифікації імен є використання у процесі ідентифікації *рекурсії*. Замість того, щоб повертати процедурі ідентифікації імен клієнта проміжні результати, під час *рекурсивної ідентифікації імені (recursive name resolution)* сервер імен передає ці результати наступному виявленому серверу імен.

Приклад. Коли кореневий сервер виявляє адресу сервера імен, реалізованого на вузлі з іменем *nl*, то він вимагає від сервера імен ідентифікувати шлях *nl:<vu, cs, ftp, pub, globe, index.txt>*, який використовує й далі рекурсивну ідентифікацію імен, ідентифікує шлях цілком і повертає файл *index.txt* кореневому серверу, який, у свою чергу, передасть його процедурі ідентифікації імен клієнта.

Рекурсивну ідентифікацію імен ілюструє рис. 5.3. Як і у разі ітеративної ідентифікації імен, останній крок ідентифікації, а саме контакт із FTP-сервером і запит на передачу файлу, відбувається на клієнті у вигляді окремого процесу.

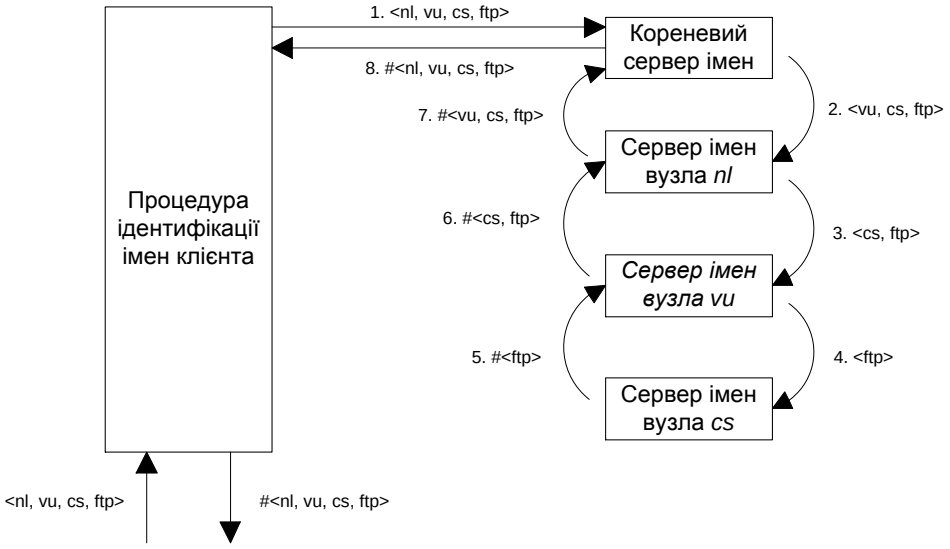


Рис. 5.3. Принцип рекурсивної ідентифікації імен

Основний недолік рекурсивної ідентифікації імен полягає в тому, що до продуктивності кожного із серверів імен ставляться підвищені вимоги. Сервер імен повинен мати змогу здійснити повну ідентифікацію шляху, хоча він міг би робити це у кооперації з іншими серверами імен. Це додаткове навантаження зазвичай настільки високе, що сервери імен глобального рівня підтримують лише *ітеративну ідентифікацію імен*.

У рекурсивної ідентифікації імен є *дві важливі переваги:*

– кешування результатів порівняно з ітеративною ідентифікацією імен є більш ефективним;

– зниження витрат на взаємодію.

Ці переваги пояснюються тим, що процедура ідентифікації імен клієнта отримує лише шляхи до вузлів глобального або адміністративного рівня. Для ідентифікації частин шляху до вузлів, які стосуються рівня керування, клієнт окремо зв'язується із сервером імен, адресу якого було повернуто йому процедурою ідентифікації імен.

Перша перевага рекурсивної ідентифікації імен полягає в тому, що вона дозволяє кожному серверу імен послідовно одержувати адресу кожного із серверів, відповідальних за вузли нижнього рівня, у результаті чого можна успішно застосувати кешування для підвищення продуктивності.

Приклад. Припустімо, кореневий сервер вимагає ідентифікувати шлях *root:<nl, vu, cs, ftp>*, у результаті чого він може одержати адресу сервера імен, який реалізує вузол, що відповідає цьому шляху. Щоб установити шлях до цього вузла, сервер імен вузла *nl* має знайти адресу сервера імен вузла *vu*, який, у свою чергу, має знайти адресу сервера імен, який відповідає за вузол *cs*. Зміни у вузлах глобального й адміністративного рівнів відбуваються нечасто, тому кореневий сервер імен може успішно кешувати отриману адресу. Більше того, оскільки в результаті рекурсії також будуть повернуті адреси серверів імен, які відповідають за реалізацію вузлів *vu* і *nl*, вони можуть кешуватися на цих серверах.

Таким чином, результати проміжного пошуку імен можна кешувати й отримувати з кешу. Якщо сервер вузла *nl* знайде адресу сервера вузла *vu*, то її може бути повернуто кореневому серверу в процесі повернення сервером *nl* результатів пошуку за вихідним іменем.

Перевага такого підходу полягає в тому, що операції пошуку стають дуже ефективними, зокрема якщо пізніше інший клієнт вимагає ідентифікувати шлях *root:<nl, vu, cs, flits>*, то це ім'я надійде в кореневий вузол, звідти негайно буде передано серверу імен вузла *cs*, до якого й надійде запит на ідентифікацію залишку шляху *cs:<flits>*.

У разі ітеративної ідентифікації імен кешування за необхідності обмежується *процедурою ідентифікації імен клієнта*. Якщо клієнт *A* запитує ідентифікацію деякого імені, а потім клієнт *B* вимагає ідентифікувати те саме ім'я, ідентифікація імені потребує повторного проходження через ті самі сервери, через які вже проходила ідентифікація для клієнта *A*. З метою досяг-

нення компромісу багато організацій використовують локальні проміжні сервери імен, спільні для всіх клієнтів. Локальні сервери імен обробляють усі запити на імена й кешують їх результати. Такі проміжні сервери зручні у керуванні, тому що у разі такої організації лише певний сервер має знати, де перебуває кореневий сервер імен, іншим машинам ця інформація не потрібна.

Друга перевага рекурсивної ідентифікації імен полягає в тому, що вона часто економніша з погляду взаємодії.

Приклад. Розглянемо ідентифікацію шляху *root:<nl, vu, cs, ftp>*, припускаючи, що клієнт перебуває в Сан-Франциско та знає адресу сервера вузла *nl*, у разі рекурсивної ідентифікації імен утвориться зв'язок між хостом клієнта в Сан-Франциско й сервером у Нідерландах, який позначено на рис. 5.4 як *R1*. Потрібно встановити зв'язок між сервером *nl* і сервером імен університету *Vrije* в університетському містечку в Амстердамі (Нідерланди). Цей зв'язок позначено як *R2*. Далі необхідний зв'язок між сервером *vu* і сервером імен факультету комп'ютерних дисциплін (Computer Science Department) *cs*, цей зв'язок позначено як *R3*. Відповідь пройде тими самими шляхами, але у зворотному напрямку. Зрозуміло, що витрати на взаємодію залежатимуть від обміну повідомленнями між хостом клієнта й сервером *nl*.

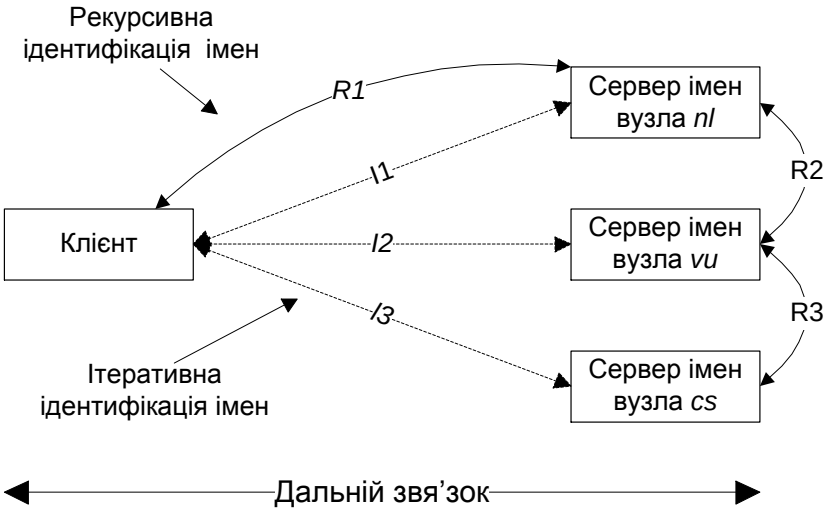


Рис. 5.4. Порівняння між рекурсивною й ітеративною ідентифікацією імен залежно від витрат на взаємодію

У разі ітеративної ідентифікації імен хост клієнта зв'язується окремо із серверами *nl, vu і cs*, у результаті чого загальні витрати будуть майже у три рази вищими, ніж у разі рекурсивної ідентифікації. Стрілки на рис. 5.4 з мітками *l1, l2* і *l3* позначають взаємодію у разі ітеративної ідентифікації імен.

Система доменних імен. Однією з найбільших на сьогодні розподілених служб іменування є **система доменних імен** Internet (Domain Name System, DNS), яку використовують передусім для пошуку адрес хостів і поштових серверів. Розглянемо організацію простору імен системи DNS й інформації, що зберігається на її вузлах, а також наявну реалізацію DNS.

Доменна система імен – розподілена система перетворення імені хоста (комп'ютера або іншого мережного пристрою) в IP-адресу.

Кожен комп'ютер в Internet має *власну унікальну адресу* – число, яке складається із чотирьох байтів. Оскільки запам'ятовувати десятки або навіть сотні достатньо важко, то всі (чи майже всі) *машини мають імена*, запам'ятати які (особливо якщо знати правила утворення імен) значно легше.

Уся система імен в Internet ієрархічна. Ієрархія потрібна, щоб не підтримувати одне централізоване джерело інформації стосовно імен, а розподілити зберігання імен хостів відповідно до місць їх розташування.

Повне доменне (domain) *ім'я машини* (Fully Qualified Domain Name, FQDN) можна поділити на дві частини – *ім'я області-домена* та *власне ім'я машини.*

Приклад. Якщо *m30.ziet.zhitomir.ua* – повне доменне ім'я машини з іменем *m30*, то *ziet.zhitomir.ua* – домен, у якому перебуває машина.

За порядок у доменах відповідає певний комп'ютер, користувачі–адміністратори якого слідкують за тим, щоб не було різних машин з однаковими IP-адресами.

Приклад. Відповідальність за область-домен *ziet.zhitomir.ua* покладено на машину *alpha.ziet.zhitomir.ua.* Ця функція делегується зверху вниз від машини *ns.lucky.net*, яка відповідає за домен *zhitomir.ua*. У свою чергу, відповідальність за область *ua* делеговано машині зверху кореневим сервером (root server).

Систему імен можна уявити у вигляді перевернутого дерева. Список імен доменів верхнього рівня побудовано як список географічних областей, який відповідає двобуквеним ISO-кодам країн і його можна знайти на WWW–сервері ISOC (http://www.isoc.org).

Необхідно розрізняти *доменне ім'я* та *поштову адресу*, в якій має бути знак «@», названий *commercial «at» sign*, що поділяє поштову адресу *на доменне ім'я* та *ім'я поштової скриньки.*

Коли мережа Internet була невеликою, таблиці відповідності імен та адрес зберігалися у звичайному текстовому файлі, який періодичн розсилався всім Internet-вузлам електронною поштою. Після того, як кількість машин значно збільшилася, така схема актуалізації таблиці відповідності імен та адрес перестала ефективно працювати, і програмісти університету штату Каліфорнія в Берклі спроектували та написали програму BIND (Berkeley Internet Name Domain), яка відповідає на запити машин користувачів, які стосуються імен та IP-адрес.

Служба імен DNS – це розподілена база даних доволі простої структури, яка має кілька таблиць, у яких записано таки відомості:

- яку IP-адресу має машина з певним іменем;
- яке ім'я має машина з визначеною адресою;
- що це за комп'ютер і яка операційна система встановлена на ньому;
- куди потрібно надсилати електронну пошту для користувачів кожної з машин;
- які псевдоніми є у певної машини.

Приклад. Розглянемо випадок, коли користувач надсилає пошту з машини *polesye.zhitomir.ua* користувачу за адресою *rozhik@ziet.zhitomir.ua*. Під час установлення на машину протоколів TCP/IP системний адміністратор зазначає IP-адресу комп'ютера – найближчого сервера імен. Поштова програма надсилає йому запит: «Куди надсилати пошту для *ziet.zhitomir.ua*». Якщо найближчий сервер не може відповісти, то він, у свою чергу, надсилає запит до сервера верхнього рівня. Нарешті, система визначає, що всю пошту для області *ziet.zhitomir.ua* необхідно відправляти на машину *alpha.ziet.zhitomir.ua* або *relay2.lucky.net*. Відповіді одночасно містять ще адресу *ziet.zhitomir.ua* машини *alpha*. Поштова програма зв'язується із цим комп'ютером (використовуючи не ім'я, а адресу) та передає йому пошту. Всі ці переговори та відправлення пошти відбуваються протягом кількох секунд, і користувач не помічає цього. Якщо машина з адресою *ziet.zhitomir.ua* недоступна, то пошта на час, коли неможливо зв'язатися з машиною за адресою *ziet.zhitomir.ua* (наприклад, під час профілактики каналу зв'язку). чекає на машині *relay2.lucky.net* своєї черги на пересилання.

Наведений приклад є характерним для поведінки Internet-програм. Поштові програми надсилають багато запитів службі DNS, алгоритми їх обслуговування складні. Здебільшого програми користувачів намагаються дізнатися лише IP-адресу певної машини. Зрозуміло, що для функціонування сис-

теми імен використовують багато винятків, правил та особливостей. Більш докладно з ними можна ознайомитися в описах стандартів Internet або у спеціальній літературі.

Простір імен DNS ієрархічно організовано у вигляді *дерева з коренем*. **Мітка** являє собою незалежний від регістра рядок алфавітно-цифрових символів. Максимальна довжина мітки – 63 символи, повна довжина шляху обмежена 255 символами. **Строкове подання шляху** складається *зі списку міток*, починаючи справа, розділених точками. **Корінь** подано *точкою*.

Приклад. Шлях *root:<nl, vu, cs, flits>* подається рядком *flits.cs.vu.nl*., включаючи точку праворуч, яка означає корінь. Зазвичай для зручності читання ця точка буде пропускатися, тому користувач бачить рядок *flits.cs.vu.nl*, без точки.

Оскільки *кожний з вузлів* у просторі імен DNS має *одне вхідне ребро* (за винятком кореня, який вхідних ребер не має), *мітка ребра, що входить у вузол,* використовується також *як ім'я цього вузла*. Піддерево називають **доменом** (*domain*), а шлях до кореневого вузла домену – **доменним іменем** (*domain name*), яке, як і шлях, може бути *абсолютним* або *відносним.*

Вміст вузла комплектується *з набору записів про ресурси* (*resource records*). Розрізняють інші типи записів про ресурси, найбільш поширені з яких подано в табл. 5.1.

Таблиця 5.1. Найбільш поширені типи записів про ресурси

Тип запису	Сутність запису	Опис
SOA	Зона	Інформація про відповідну зону
A	Хост	IP-адреса хоста, на якому встановлено вузол
MX	Домен	Поштовий сервер, що обробляє поштові адреси вузла
SRV	Домен	Сервер, що надає деяку службу
NS	Зона	Сервер імен, що відповідає за певну зону
CNAME	Вузол	Символічне посилання на первинне ім'я відповідного вузла
PTR	Хост	Канонічне ім'я хоста
HINFO	Хост	Інформація про хост, на якому встановлено вузол
TXT	Будь-яка сутність	Додаткова інформація, що залежить від сутності

Вузол у просторі імен DNS часто представлений одночасно декількома сутностями. Так, доменне ім'я *vu.nl* використовується для представлення домену і зони, тому що домен реалізовано в декількох зонах.

Запис про ресурс SOA (*start of authority* – початок повноважень) містить таку інформацію, як *поштова адреса системного адміністратора*, відповідального за зазначену зону, *ім'я хоста*, на якому перебувають дані про зону.

Запис *A* (*address* – адреса) – це *окремий хост в Internet*, що містить IP–адресу хоста, який використовується у процесі взаємодії. Якщо хост має кілька IP-адрес, наприклад у машинах із множинною адресацією, то вузол міститиме запис *A* для кожної адреси.

Запис MX (*mail exchange* – поштовий обмін) є символічним посиланням на вузол, що представляє *поштовий сервер*.

Приклад. Нехай вузол, який являє собою домен *cs.vu.nl*, має запис MX, що містить ім'я *zephyr.cs.vu.nl*, яке стосується поштового сервера. Сервер оброблятиме всі адреси для вхідних поштових повідомлень користувачів домену *cs.vu.nl*. Вузол може мати кілька записів MX.

Записи SRV аналогічні записам MX, вони містять ім'я сервера **конкретної служби**, яка визначається за її іменем та іменем протоколу.

Приклад. Так, web-сервер домену *cs.vu.nl* може бути іменований за допомогою такого запису SRV, як *http.tcp.cs.vu.nl*, який може надсилатися потім і на реальне ім'я сервера *(soling.cs.vu.nl).*

Вузли, які представляють зону, містять один або більше **записів NS** (*name server* – сервер імен), ці записи, як і запис MX, містять *ім'я сервера імен*, який реалізує зону, що є вузлом. Кожний вузол простору імен може мати запис NS, який посилається на сервер імен, що його реалізує. Однак реалізація простору імен DNS така, що необхідним є запис NS тільки для вузлів, які представляють зону.

DNS має *псевдоніми* для *канонічних імен (canonical names - CNAME)*, оскільки кожний хост має канонічне, або первинне, ім'я. Псевдонім реалізується за допомогою вузла, який зберігає запис *CNAME*, у якому міститься канонічне ім'я хоста. Ім'я вузла, на якому зберігається цей запис, збігається з іменем символічного посилання, як показано на рис. 5.3.

DNS підтримує *зворотне відображення* IP-адрес на імена хостів за допомогою записів *PTR* (*pointer* – вказівник). Для того, щоб виконати пошук серед імен хостів, коли відома тільки IP-адреса, DNS підтримує домен *in-addr.arpa*, що містить вузли, які представляють хости Internet і впорядковані відповідно до IP-адрес цих хостів.

Приклад. Так, для хоста *www.cs.vu.nl*, що має IP-адресу *130.37.24.11*, DNS створює вузол з іменем *11.24.37.130.in-addr.arpa*, яке використовується для зберігання в записі PTR канонічного імені цього хоста (а саме, *soling.cs.vu.nl*).

Запис HINFO (*host info* – інформація про хост) використовують для збереження додаткової інформації про хост, зокрема про тип машини та операційну систему.

Записи TXT (*text* – текст) використовують для збереження будь-якої додаткової інформації, яку може зберегти користувач про сутності, представлені вузлом.

Реалізація DNS. Простір імен DNS можна поділити на **глобальний** і **адміністративний рівні**. **Рівень керування**, який зазвичай охоплює локальні файлові системи, формально не є частиною системи DNS і не контролюється нею.

Кожна зона реалізована у вигляді сервера імен. Для кращої доступності ці сервери постійно піддаються **віртуальній реплікації**. Зміни в зонах, які зазвичай обробляються **первинним сервером імен**, вносяться через модифікацію баз даних DNS локально. **Вторинні сервери** імен не мають прямого доступу до баз даних, вони надсилають запити первинному серверу, який у відповідь передає їм свій вміст. Цей процес називають **перенесенням зон** (*zone transfer*).

База даних DNS реалізується у вигляді набору (невеликого) файлів, серед яких найбільш важливий містить усі записи про ресурси з усіх вузлів відповідної зони. Такий підхід дозволяє легко ідентифікувати вузли за їх доменними іменами, через що ідентифікатор вузла стає (неявно) індексом файлу. В табл. 5.2 наведено частину файлу, в якому міститься основна інформація про домен *cs.vu.nl*. Файл демонструє вміст восьми різних вузлів, які являють собою частину домену *cs.vu.nl*, кожен вузол ідентифікується власним доменним ім’ям.

Таблиця 5.2. Частина бази даних DNS для зони *cs.vu.nl*

№ з\п	Ім'я	Тип запису	Значення запису
1.	*cs.vu.nl*	SOA	*star* (1999121502.7200.3600.2419200.86400)
2.	*cs.vu.nl*	NS	*star.cs.vu.nl*
3.	*cs.vu.nl*	NS	*top.cs.vu.nl*
4.	*cs.vu.nl*	NS	*solo.cs.vu.nl*
5.	*cs.vu.nl*	TXT	*"Vrije Universiteit – Math. & Comp. Sc."*
6.	*cs.vu.nl*	MX	*1 zephyr .cs.vu.nl*
7.	*cs.vu.nl*	MX	*2 tornado .cs.vu.nl*
8.	*cs.vu.nl*	MX	*3 star .cs.vu.nl*
9.	*star.cs.vu.nl*	HINFO	*Sun Unix*
10.	*star.cs.vu.nl*	MX	*1 star .cs.vu.nl*
11.	*star.cs.vu.nl*	MX	*10 zephyr.cs.vu.nl*
12.	*star.cs.vu.nl*	A	130.37.24.6
13.	*star.cs.vu.nl*	A	192.31.231.42
14.	*zephyr.cs.vu.nl*	HINFO	*Sun Unix*
15.	*zephyr.cs.vu.nl*	MX	*1 zephyr .cs.vu.nl*
16.	*zephyr.cs.vu.nl*	MX	*2 tornado .cs.vu.nl*
17.	*zephyr.cs.vu.nl*	A	192.31.231.66
18.	*www.cs.vu.nl*	CNAME	*soling.cs.vu.nl*
19.	*ftp.cs.vu.nl*	CNAME	*soling.cs.vu.nl*
20.	*soling.cs.vu.nl*	HINFO	*Sun Unix*
21.	*soling.cs.vu.nl*	MX	*1 soling.cs.vu.nl*
22.	*soling.cs.vu.nl*	MX	*10zephyr.cs.vu.nl*
23.	*soling.cs.vu.nl*	A	130.37.24.11
24.	*laser.cs.vu.nl*	HINFO	*PC MS-DOS*
25.	*laser.cs.vu.nl*	A	130.37.30.32
26.	*vucs-das.cs.vu.nl*	PTR	*0.26.37.130. in-addr. arpa*
27.	*vucs-das.cs.vu.nl*	A	130.37.26.0

Вузол *cs.vu.nl*, наприклад, являє собою як домен, так і зону. Його запис про ресурси SOA містить спеціальну інформацію про достовірність цього файлу. В цій зоні є три сервери імен, канонічні імена їх хостів містяться в записах NS. Запис TXT використовується для зберігання додаткової

інформації та не може бути оброблений серверами імен автоматично. Є три поштових сервери, що обробляють пошту, адресовану користувачам цього домену. Число перед іменем поштового сервера вказує на пріоритет вибору. Сервер, що відправляє пошту, спочатку намагається встановити контакт із приймаючим сервером з найменшим номером, тобто *zephyr.cs.vu.nl*.

Сервери імен важливі для будь-якої служби імен. Хост *star.cs.vu.nl* виконує обов'язки сервера імен цієї зони, тому йому властива додаткова стійкість, створювана двома незалежними мережними інтерфейсами, кожний з яких представлений окремим записом про ресурси *A*. Таким чином, збій зв'язку в мережі може відбутися непомітно.

Наступні чотири рядки (14–17) надають необхідну інформацію з поштового сервера, який архівується на інший поштовий сервер, шлях до якого – *tornado.cs.vu.nl*.

Наступні шість рядків (18–23) демонструють типову конфігурацію web-сервера факультету, в якому на одній з машин реалізовано FTP-сервер з іменем *soling.cs.vu.nl*. Підтримка системи значно полегшується під час виконання обох серверів на одній машині та використання цієї машини тільки для служб Internet. Так, обидва сервери можуть працювати з одним представленням файлової системи, а для підвищення ефективності функціонування частину файлової системи можна реалізувати на сервері *soling.cs.vu.nl*. Такий підхід часто застосовують для служб WWW і FTP.

Наступні два рядки (24-25) містять інформацію про один з лазерних принтерів, з'єднаних із локальною мережею. Останні два рядки (26-27) демонструють зворотне відображення адрес у канонічні імена, за якого ім'я суперкомп'ютера відділу може бути знайдено за його адресою в домені *in-addr.arpa.*

Оскільки домен *cs.vu.nl* реалізовано в одній зоні, в табл. 5.2 немає посилань на інші зони. Спосіб створення посилань на піддомени, які реалізуються в іншій зоні, ілюструє табл. 5.3. Для того, щоб створити посилання на піддомени, необхідно вказати сервер імен для піддомена, зазначивши його доменне ім'я та IP-адресу. Під час дозволу імені вузла, який не входить у домен *cs.vu.nl,* дозвіл імені продовжиться у місці, адреса якого буде зчитана з бази даних DNS для сервера імен домену *cs.vu.nl.*

Таблиця 5.3. Частина опису домену *vu.nl*, у якій міститься домен *cs.vu.nl*

Ім'я	Тип запису	Значення запису
cs.vu.nl	NS	*solo.cs.vu.nl*
solo.cs.vu.nl	A	130.37.24.1

5.1.4 Стандарт X.500

Стандарт X.500 – серія стандартів ITU-T (1993 р.) для *служби розподіленого каталогу мережі*, які надають централізовану інформацію про всі іменовані об'єкти мережі (ресурси, прикладне програмне забезпечення та користувачів, рекомендації МККТТ для каталогів). Спочатку стандарт X.500 планували для використання іменувань вузлів, адрес та поштових скриньок, передбачених стандартом X.400.

Каталоги містять статичні й рідко змінювані елементи, оскільки їх оптимізовано для дуже швидкого відгуку на запити пошуку і читання даних. Каталоги повністю структуровані: кожен елемент даних має ім'я, яке одночасно визначає положення елемента в ієрархії каталогу; кожен атрибут елемента може мати кілька значень, що нормально на відміну від звичайних баз даних.

Каталоги є дуже специфічними системами зберігання даних, їх зручно використовувати для ієрархічно скомпонованих об'єктів. Каталоги можуть бути реплікованими між декількома серверами для зручного доступу і розподілу навантаження. Текстову інформацію достатньо легко зберігати в каталогах, оскільки ієрархічна структура файлів спрощує пошук, але дані можуть подаватися у будь-якій іншій формі.

Каталоги зручно використовувати для керування призначеними для користувача акаунтами, машинами, схемами доступу, прикладним програмним забезпеченням, оскільки механізми керування найчастіше тільки зчитують дані з центрального сховища. Каталоги можна гнучко налаштувати для будь-якого рівня контролю доступу, дозволяючи обмежувати доступ до інформації за потребою, їх часто використовують для зберігання інформації про користувачів або реальні об'єкти.

Приклад. Прикладне програмне забезпечення може обробляти дані, асоційовані з людиною, зокрема реальну або мережну адресу, телефон, e-mail, рівень доступу тощо. Оскі-

льки до багатьох програм інтегровано засоби доступу до каталогів, то користувачі значно ефективніше працюють із програмним забезпеченням, призначеним для спільної роботи. Дуже зручно використовувати каталоги для централізованого керування мережею, оскільки дані про користувачів, парк машин і запущені сервіси зосереджені в одному місці та легко адмініструються.

Виходячи з вищесказаного, можна назвати ті області, в яких використання директорій буде максимально ефективним:

Спрощене керування мережею:

- централізоване керування інформацією про людей;
- централізоване керування мережними об’єктами та їх конфігурацією;
- централізоване керування акаунтами користувачів;
- зменшення вартості підтримки апаратно-програмної інфраструктури за рахунок централізації.

Уніфікований доступ до мережних ресурсів:

- уніфікований підхід до іменування;
- можливість єдиного входу для доступу до всіх ресурсів.

Єдине місце зберігання загальної інформації:

- контактних даних;
- місця розташування мережних ресурсів;
- можливість використання для зберігання будь-якої інформації.

Покращене керування даними:

- єдине місце зберігання широко використовуваних даних;
- єдине місце керування доступом до даних;
- організація даних у єдину логічну структуру.

Допомага в організації потокового процесу роботи.

Єдиний репозиторій та пошуковий механізм для прикладного програмного забезпезпечення і сервісів.

Система DNS – приклад традиційної служби іменування, коли задані імена (можливо, ієрархічно побудовані), – ідентифікує ім’я як вузол у графі іменування й повертає вміст цього вузла у вигляді запису про ресурси. Ідентифікація імені в DNS аналогічна пошуку номера телефону в телефонній книзі.

Служба каталогів (directory service) – це особливий тип служби іменування, за якої клієнт може виконувати пошук сутності на підставі опису її властивостей, а не повного імені. Цей підхід аналогічний тому, як використовують довідник «Жовті сторінки».

Простір імен X.500. Концептуально служба каталогів X.500 містить безліч записів, які називають **елементами каталогу**, які є подібними до запису про ресурси системи DNS. Кожний запис складається з набору пар (атрибут, значення), причому кожний атрибут має асоційований з ним тип. Розрізняють атрибути з одним значенням (однозначні) і атрибути з декількома значеннями (багатозначні), які являють собою масиви або списки.

Приклад. Так, представлення кожного елемента каталогу, що визначає мережні адреси декількох основних серверів із табл. 5.2, наведено в табл. 5.4. У цьому прикладі використовується спосіб іменування, описаний у стандартах X500, відповідно до яких і обирають значення перших п'ять атрибутів. Атрибути *Organization, OrganizationUnit* описують відповідно організацію і підрозділ, що асоціюються з даними, які зберігаються в записі. Атрибути *Locality, Country* надають додаткову інформацію про те, де зберігається елемент. Атрибут *CommonName* часто використовується як ім'я для ідентифікації елемента в обмеженій частині каталог. Так, імені «*Main Server*» може бути достатньо для того, щоб знайти елемент із цього прикладу, бо воно має відповідні значення інших чотирьох атрибутів – *Locality, Country, Organization, OrganizationUnit*. Лише атрибут *Mail_Servers* має декілька значень, а решта – однозначні.

Таблиця 5.4. Приклад елемента каталогу X.500

№ з\п	Атрибут	Абревіатура	Значення
1.	Country	C	NL
2.	Locality	L	Amsterdam
3.	Organization	O	Vrije Universiteit
4.	OrganizationUnit	OU	Math. & Comp. Sc.
5.	CommonName	CN	Main server
6.	Mail_Servers	_	130.37.24.6 192.31.231.42 192.31.231.66
7.	FTP_Server	–	130.37.24.11
8.	WWW_Server	–	130.37.24.11

Набір усіх елементів каталогу служби каталогів X.500 називають **інформаційною базою каталогу** (*Directory Information Base, DIB*). Кожний запис

у DIB має *унікальне ім'я*, щоб його можна було знайти. Глобальним унікальне ім'я стає завдяки використанню послідовності атрибутів іменування кожного запису. Кожний атрибут іменування називають *відносно визначеним іменем (Relative Distinguished Name, RDN).*

Приклад. У попередньому прикладі (табл. 5.4) перші п'ять атрибутів – це всі атрибути іменування. Якщо застосувати стандартні скорочення для подання атрибутів іменування X.500, атрибути *Country*, *Organization* і *OrganizationalUnit* можна використовувати для формування глобального унікального імені */C=NL/0=Vrije Universiteit/OU=Math. & Comp.Sc*. Це ім'я аналогічно імені *nl.vu.cs* у системі DNS.

Як і в DNS, використання глобально унікальних імен, утворених послідовним упорядкуванням імен RDN, зумовить ієрархію наборів елементів каталогу, який називають **інформаційним деревом каталогів** (*Directory Information Tree, DIT*). За допомогою DIT утворюють *граф іменування* служби каталогів X.500, у якому кожний вузол є елементом каталогу та може мати кілька дочірніх вузлів, для яких він буде батьком (рис. 5.5).

Вузол виконує роль батька декількох інших елементів каталогу, які мають додатковий атрибут іменування *Host_Name*, використовуваний як RDN. Ці сутності можна застосувати для представлення хостів.

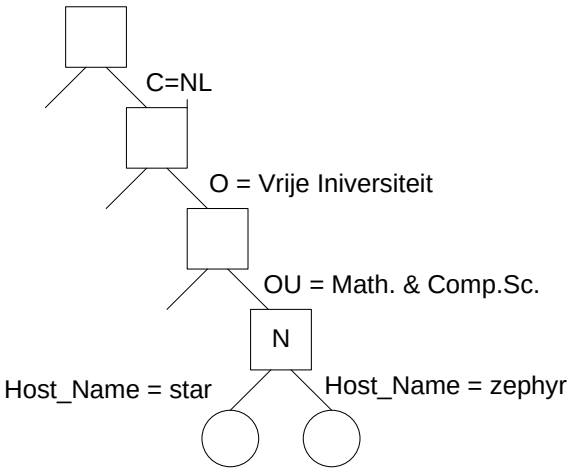


Рис. 5.5. Частина інформаційного дерева каталогів

Реалізація X.500. Реалізація служби каталогів X.500 виконується так само, як і реалізація служби імен, зокрема DNS, але підтримує більше операцій пошуку. В разі операцій з великим каталогом дерево DIT зазвичай розділяється на частини та їх розносять по декільком серверам, які відповідно до

X.500 називають **агентами служби каталогів** (*Directory Service Agents*, DSA). Кожна частина дерева DIT відповідає зоні в DNS, так само кожний агент DSA поводиться аналогічно зі звичайним сервером імен, але додатково він реалізує стандартні функції для служби каталогів, наприклад розширені операції пошуку.

Клієнти служби каталогів X.500 представлені **агентами користувачів каталогу** (*Directory User Agents, DUA*), поведінка яких подібна до процедури ідентифікації імен з традиційної служби іменування, оскільки DUA обмінюються інформацією з DSA відповідно до стандартного протоколу доступу.

Відмінність реалізації X.500 від реалізації DNS полягає у наявності *механізмів пошуку* в базі DIB, зокрема пошуку *елемента каталогу за заданим набором критеріїв*, який містить атрибути елементів пошуку.

Приклад. Припустімо, необхідно одержати список усіх головних серверів університету, який можна повернути, використовуючи операцію пошуку *answer = search("&(C-NL)(0-Vnje Universiteit)(OU=*)(CN=Main server)").*

Місцем пошуку головних серверів є організація *Vrije Universitelt* у країні, яку позначено *NL*, при цьому не важливий конкретний відділ цієї організації, однак кожний результат, який повертається, повинен мати значення атрибута *CN*, що становить *Main server*.

Слід відзначити, що такий пошук у службі каталогів – зазвичай достатньо дорога операція. Так, щоб знайти всі головні сервери університету, необхідно визначити всі сутності в кожному відділі й зібрати результати пошуку в одну відповідь, тобто щоб одержати відповідь, необхідно перебрати значну кількість листових вузлів дерева DIT, а на практиці й значну кількість агентів DSA. Для запобігання цьому служби імен можуть бути реалізовані за допомогою операції пошуку, що потребує доступу тільки до одного листового вузла.

Система X.500 стандартизована як і протоколи OSI, отримати доступ до її каталогу відповідно до офіційних правил не просто. Щоб пристосувати службу каталогів X.500 до Internet, було створено **спрощений протокол доступу до каталогів** (*Lightweight Directory Access Protocol, LDAP*), який є протоколом прикладного рівня, що реалізується поверх TCP. Така реалізація спрощує його використання порівняно з офіційним протоколом доступу OSI. Крім того, параметри операцій пошуку й відновлення можуть бути передані у вигляді рядків, а

роздільного кодування, необхідного у протоколі OSI, не потрібно. Протокол LDAP став стандартом де-факто для служб каталогів у Internet.

5.2 Розміщення мобільних сутностей

Служби імен, які розглянуто, використовують передусім для іменованих сутностей, що мають постійне місце розташування. За своєю природою традиційні системи іменування погано підтримують відображення імені на адресу, якщо вона регулярно змінюється, як це відбувається у мобільних сутностях.

5.2.1 Іменування й локалізація сутностей

Сутності йменують для того, щоб мати можливість їх знайти й одержати до них *доступ*. Розрізняють такі типи імен: *імена*, *зручні для сприйняття*, *ідентифікатори* й *адреси*. Оскільки розподілені системи створюють для людей і для доступу до сутності, то необхідно знати адресу сутності, тому фактично всі системи іменування підтримують відображення зручних для сприйняття імен сутностей у їх адреси.

Для ефективної реалізації повномасштабного простору імен, такого як у DNS, зручно поділити простір імен на три рівні: *глобальний*, *адміністративний та керування*. **Глобальний і адміністративний** рівні характеризуються тим, що імена змінюються нечасто, точніше, вміст вузлів цих частин простору імен порівняно постійний, унаслідок чого реплікація й кешування здатні підвищити ефективність реалізації.

Вміст вузлів **рівня керування** часто змінюється, тому продуктивність операцій пошуку й відновлення стає особливо важливою. На практиці вимоги до продуктивності можна задовольнити за рахунок реалізації вузлів на локальних високопродуктивних серверах імен.

Приклад. Розглянемо, як подібний підхід можна використовувати для реалізації великомасштабної системи іменування. Насамперед виконаємо пошук адреси віддаленого хоста *ftp.cs.vu.nl*. Якщо вважати вміст вузлів глобального й адміністративного рівнів стабільним, то клієнт зможе знайти адресу сервера імен домену *cs.vu.nl* у локальному кеші за допомогою лише одного запиту до сервера імен для пошуку адреси *ftp.cs.vu.nl*.

Якщо адресу *ftp.cs.vu.nl* змінено, наприклад, через перенесення FTP-сервера на іншу машину, то за умови, що сервер буде залишатися на машині, яка входить у домен *cs.vu.nl*, відновлення можна виконати швидко, тоді змінюється лише база даних DNS сервера імен *cs.vu.nl*, а пошук буде такий же ефективний, як і раніше.

Отже, якщо вузли глобального й адміністративного рівнів змінюються рідко, а зміни зазвичай обмежуються одним сервером імен, то системи іменування, зокрема DNS, достатньо ефективні.

Приклад. Якщо сервер *ftp.cs.vu.nl* перенести на машину з іменем *ftp.cs.umsa.edu.au*, розташовану в іншому домені, то передусім варто звернути увагу на те, що ім'я *ftp.cs.vu.nl* краще було б не змінювати, оскільки багато прикладних програм і користувачів, вірогідно, мали на нього лише символічне посилання, тобто це ім'я використовують як ідентифікатор, а його зміна зробить усі посилання на нього неправильними.

Існує два основних рішення проблеми перенесення сервера в інший домен: 1. Записати адресу нової машини в базі даних DNS для *cs.vu.nl*; 2. Записати ім'я нової машини, а не її адресу, зазначивши *fip.cs.vu.nl* у символічному посиланні. Обидва рішення мають суттєві недоліки.

Спочатку розглянемо запис адреси нової машини (перший підхід, який не змінює алгоритму операції пошуку). Однак, якщо сервер *ftp.cs.vu.nl* доведеться ще раз переносити на нову машину, то необхідно оновлювати і його елемент бази даних DNS у *cs.vu.nl*. Важливо відзначити, що оновлення виконується не довше локальної операції, але на практиці може потребувати сотень мілісекунд. Недоліком такого підходу є порушення припущення про те, що операції у вузлах рівня керування ефективні.

Основний недолік використання символічних посилань (другий підхід) полягає в тому, що втрачається ефективність операцій пошуку, кожна з яких виконується за два кроки: пошук імені нової машини; пошук адреси, що відповідає цьому імені.

Якщо сервер *ftp.cs.vu.nl* знову переміститься, скажімо, на адресу *ftp.cs.berke-ley.edu*, то можна здійснити операцію локального відновлення, помістивши ім'я *ftp.cs.unisa.edu.au* у символічне посилання на *ftp.cs.berkeley.edu* і зберігши для *cs.vu.nl* елемент у базі даних DNS таким, як він є. Недоліком другого підходу є необхідність додавання ще одного кроку до операції пошуку.

Для сутностей з високою мобільністю ситуація є значно гіршою. Щоразу під час переносення сутності доводиться виконувати нелокальну операцію внесення змін або додавати ще один крок до операції пошуку.

Іншим суттєвим недоліком розглянутих підходів є те, що *ім'я*, наприклад *ftp.cs.vu.nl*, не має змінюватися, тому надзвичайно важливо обирати імена, які не доведеться змінювати протягом усього часу існування тієї сутності, яку

вони ідентифікують. Більше того, таке ім'я не можна використовувати для яких-небудь інших сутностей. На практиці підібрати такі імена, особливо для довгоживучих сутностей, складно, особливо у системі іменування в World Wide Web. Багато сайтів відомі під різними іменами, і всі ці імена залишаються правильними, тобто завжди посилаються на одну сутність, навіть у разі мобільності. У зв'язку із цим традиційні служби імен, такі як DNS, не можуть працювати ефективно з мобільними сутностями, тому для цього необхідно знайти інше рішення. Суттю проблеми є те, що традиційні служби іменування підтримують пряме відображення зручних для сприйняття людиною імен в адреси сутностей. Щоразу, коли ім'я або адреса змінюються, доводиться змінювати й відображення, як показано на рис. 5.6, *а*.

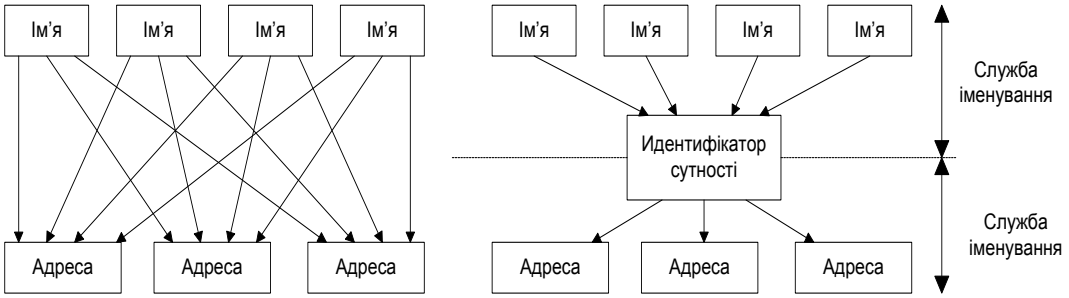


Рис. 5.6. Пряме однорівневе відображення імен в адреси (*а*); дворівневе відображення з використанням ідентифікаторів (*б*)

Найкраще рішення зазначеної проблеми полягає у відокремленні іменування сутностей від їх розміщення за допомогою введення ідентифікаторів, як показано на рис. 5.6, б. Нагадаємо, що ідентифікатори ніколи не змінюються, кожна сутність має тільки один ідентифікатор, який не може ідентифікувати іншу сутність, а також не призначений для сприйняття людиною, тобто оптимізований винятково для машинної обробки.

Під час пошуку сутності засобами служби іменування вона повертає ідентифікатор, який може бути збережений на локальній машині на будь-який необхідний термін, оскільки він не може ані почати вказувати на іншу сутність, ані змінитися, причому не важливо, під яким іменем його збережено. Коли цей ідентифікатор знадобиться знову, його можна буде взяти з локальної машини, не виконуючи пошуку засобами служби іменування.

Розміщення сутності визначається за допомогою окремої **служби локалізації** *(location service)*, яка використовує як вихідні дані *ідентифікатор* і повертає *поточну адресу* відповідної сутності. Якщо є кілька копій сутності, то буде повернуто кілька адрес.

5.2.2 Реалізація ефективної служби локалізації

Для реалізації ефективної служби локалізації наявні два простих рішення щодо локалізації сутностей, які можна застосувати лише в локальних мережах, де вони зазвичай успішно працюють.

Широкомовне і групове розсилання. Розглянемо розподілену систему, побудовану на основі комп'ютерної мережі, яка надає ефективні механізми широкомовного розсилання, які зазвичай надаються в локальній мережі, де всі машини приєднані до одного кабелю. Безпроводові локальні мережі також використовують ці механізми.

Локалізація сутності в такому середовищі нескладна: повідомлення, які містить ідентифікатор сутності, широкомовним розсиланням передається кожній машині, й кожна з них відповідає на цей запит перевіркою, чи не на ній розміщена ця сутність. Ті машини, які можуть надати *точку входу* до шуканої сутності, надсилають відповідне повідомлення, яке містить *адресу точки входу*.

Цей принцип використано у протоколі ідентифікації адрес *(Address Resolution Protocol, ARP)* Internet для пошуку канальної адреси машини, для якої відома тільки IP-адреса. Фактично, машина, що надсилає широкомовний пакет у локальну мережу, запитує, хто має зазначену IP-адресу. Коли машина приймає таке повідомлення, одержувач перевіряє, чи не знає він цю IP-адресу, якщо це так, вона надсилає відповідний пакет, який містить, наприклад, його адресу в мережі Ethernet.

Широкомовне розсилання зі збільшенням мережі втрачає ефективність, причиною цього є втрати пропускної здатності мережі на пересилання повідомлень, але це не єдина проблема. Більш важливо те, що безліч хостів змушені переривати свою роботу через запит, на який вони не відповідатимуть. Одне з можливих рішень цієї проблеми – перехід *до групового розси-*

лання, під час якого запит одержує лише обмежена група хостів. Так, мережа Ethernet апаратно підтримує *групове розсилання канального рівня*.

Групове розсилання може також використовуватися для локалізації сутностей у мережах, що не комутуються. В Internet підтримується *групове розсилання мережного рівня*, за якого хостам дозволено приєднуватися до конкретної групи розсилання, що визначається *адресою групового розсилання.* Коли хост відправляє повідомлення на адресу групового розсилання, мережний рівень надає зручну службу з доставки цього повідомлення кожному із членів групи. **Адреса групового розсилання** може використовуватися як узагальнена служба локалізації для безлічі сутностей.

Приклад. Розглянемо підприємство, у якому кожний співробітник має власний мобільний комп'ютер. Коли цей комп'ютер приєднується до локальної мережі, йому динамічно надається IP-адреса.

Інший спосіб використання адреси групового розсилання – асоціювати його з реплікованою сутністю і за рахунок групового розсилання виконати пошук найближчої репліки. Під час відправлення запиту на адресу групового розсилання кожна репліка повідомляє свою поточну (звичайну) IP-адресу. Найпростіший спосіб отримати найближчу репліку – взяти ту, відповідь від якої надійшла першою. У загальному випадку алгоритм знаходження найближчої репліки є дуже складним.

Передача вказівників. Інший поширений підхід до локалізації мобільних сутностей ґрунтується *на передачі вказівників*. Коли сутність переміщується від *A* до *B*, вона зберігає посилання на своє нове місце розташування в *A*. Перевагою цього підходу є його простота: як тільки сутність локалізується за допомогою, наприклад, традиційної служби іменування, клієнт може знайти її поточну адресу, пройшовши ланцюжком переданих вказівників.

Цей принцип має такі недоліки: якщо не вживати спеціальних заходів, то ланцюжок може стати настільки довгим, що локалізація сутності коштуватиме занадто дорого; всі проміжні місця розташування в ланцюжку мають підтримувати свою частину ланцюжка так довго, як це буде необхідно; уразливість до втрати посилань, що суттєво, оскільки якщо пересилається тільки вказівник, і його, з якихось причин, буде втрачено, то неможливо буде лока-

лізувати сутність. У зв'язку з цим важливо намагатися утримувати ланцюжок коротким і гарантувати збереження вказівників, які пересилаються.

Щоб краще зрозуміти, як відбувається передача вказівників, розглянемо її в контексті розподілених об'єктів. Згідно з підходом *ланцюжків SSP* (*Stub-Scion Protocol chains, SSP chains*) кожний з вказівників, які пересилаються, реалізується у вигляді пари (*замісник, скелетон*).

У SSP замісника (*proxy*) називають **заглушкою** (*stub*), а скелетон (*skeleton*) – **паростком** (*scion*), що приводить до пари (*stub*, *scion*), від якої й утворено абревіатуру SSP.

Скелетон (тобто серверна заглушка) містить локальне посилання або на реальний об'єкт, або на замісника (тобто клієнтську заглушку) цього об'єкта. Щоб підкреслити, що скелетони виконують для віддалених посилань роль вхідних елементів, а замісники – вихідних елементів, їх позначено на рис. 5.7 опуклим п'ятикутником.

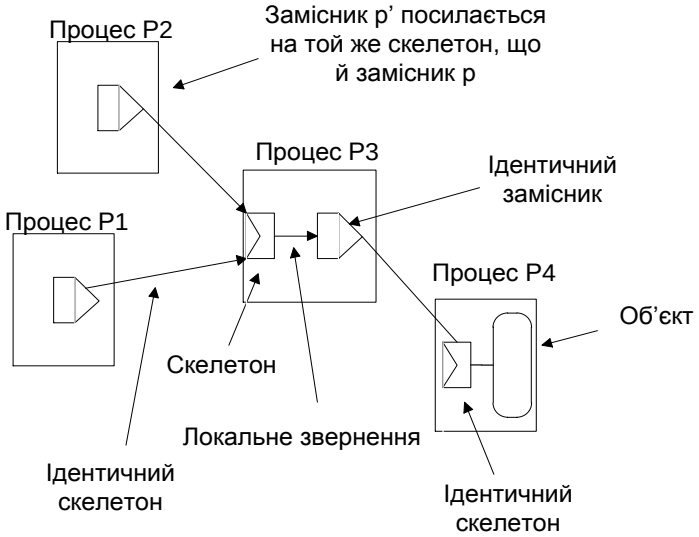


Рис. 5.7. Принцип передачі вказівників з використанням пар (замісника, скелетона)

Щоразу передаючи об'єкт з адресного простору *A* в адресний простір *B*, об'єкт залишає замість себе в адресному просторі *A* замісника, а для зв'язку із цим замісником в адресному просторі *B* установлює скелетон. Ці переміщення абсолютно не помічає клієнт, а бачить лише замісника. Від клієнта приховано, як і куди замісник передає виклики. Відзначимо, що алгоритм передачі вказів-

ників відрізняється від алгоритму пошуку адреси. Замість того, щоб шукати адресу об’єкта, передається запит клієнта до реального об’єкта ланцюжком.

Для зменшення довжини ланцюжка пар (замісник, скелетон) разом *з викликом* передається *ідентифікатор того замісника*, який ініціював цей виклик. Ідентифікатор замісника містить адресу клієнта транспортного рівня разом із числом, яке ідентифікує конкретного замісника. Коли виклик досягає об’єкта в тому місці, де він розташований у цей момент, відповідь повертається безпосередньо тому замісникові, який ініціював виклик. Поточне місце розташування об’єкта вміщується у повідомлення-відповідь, і замісник змінює той скелетон, з яким він працював, на скелетон з поточного місця розташування об’єкта. Цей принцип проілюстровано на рис. 5.8.

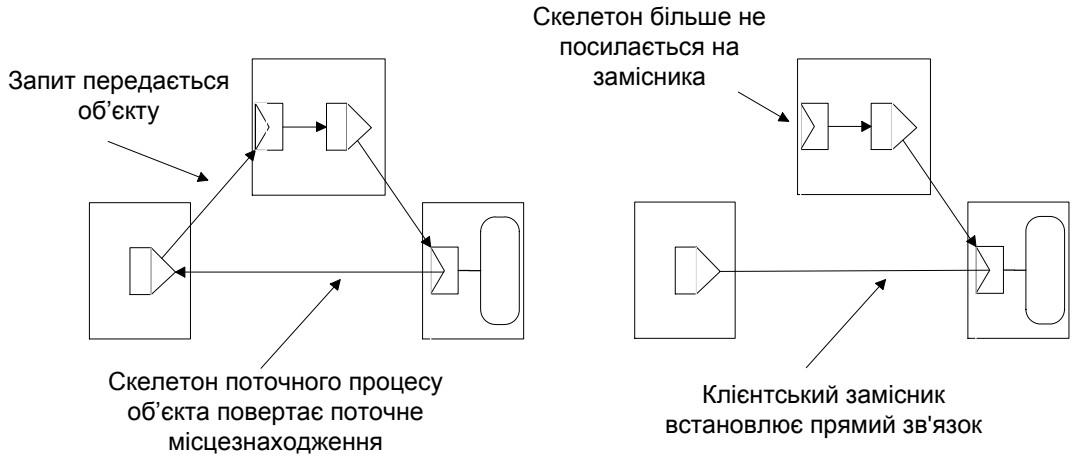


Рис. 5.8. Перенапрямлення переданого вказівника за рахунок налаштування замісника на прямий зв’язок

Надсилання відповіді замісникові, який ініціював обмін, та надсилання відповіді назад ланцюжком вказівників, які передаються, дещо відрізняються. У разі надсилання відповіді замісникові, який ініціював обмін, зв’язок установлюється швидше, оскільки залучається невелика кількість процесів, але підкоригувати можна тільки замісника, у той час як надсилання відповіді назад ланцюжком дає можливість коригувати всіх проміжних замісників. Після того як скелетон припинить посилатися на якого-небудь замісника, його можна видалити.

Посилання на об’єкти в розподілених системах можуть бути реалізовані у вигляді замісників, причому ця реалізація підходить також для пересилання вказівників.

Проблеми виникають тоді, коли процес у ланцюжку пар (замісник, скелетон) зависає, що робить інші процеси недоступними. Є декілька розв’язків цієї проблеми. Один з них – широко використовуваний – дозволити машині, на якій було створено об’єкт (її називають *базовою точкою (home location) об’єкта*), завжди зберігати посилання на його поточне місцезнаходження та обробляти так, щоб запобігти збоям. У разі обриву ланцюжка поточне місцезнаходження об’єкта запитується з базової точки об’єкта, для зміни якої можна використовувати традиційну службу імен, у якій записується поточна базова точка.

5.2.3 Підхід до локалізації на основі базової точки

Використання широкомовного розсилання й передачі вказівників створює проблеми масштабованості. Широкомовні та групові розсилання важко ефективно реалізувати у великомасштабних мережах, а довгі ланцюжки вказівників, які пересилаються, створюють проблеми з продуктивністю й чутливістю до обривів зв’язків.

Поширений підхід до підтримки мобільних сутностей у повномасштабних мережах полягає у введенні поняття *«базова точка» (home location)*, з якої відслідковується поточне місцезнаходження об’єкта. Для захисту базової точки від збоїв у мережі або процесах можна застосовувати спеціальні методики. На практиці базовою точкою зазвичай є те місце, де було створено сутність.

Підхід на основі базової точки використовується як *аварійний метод* служб локалізації, які ґрунтуються на передачі вказівників. Іншим прикладом такого підходу є схема мобільної IP-адреси, за якою кожний мобільний хост має свою фіксовану IP-адресу. Будь-який зв’язок із цією IP-адресою спершу перенапрямляється *агентові бази (home agent)* мобільного хоста в локальній мережі, яка відповідає мережній адресі та містить IP-адресу мобільного хоста. Щоразу, коли мобільний хост переміщується в іншу мережу, він запитує

тимчасову адресу для зв’язку, яку називають *контрольною* або адресою, що виходу з мережі (*care-of address*), реєструється агентом бази.

Коли агент бази одержує від мобільного хоста пакет, він перевіряє місцезнаходження хоста. Якщо хост перебуває в поточній локальній мережі, то пакет пересилається йому. Інакше він передається туди, де в цей час перебуває хост, тобто поміщається у вигляді даних в IP-пакет і пересилається на контрольну адресу. Одночасно відправник пакета повідомляється про поточне місцезнаходження хоста. Цей принцип ілюструє рис. 5.9. Відзначимо, що IP-адреса успішно використовується як ідентифікатор мобільного хоста.

Рис. 5.9. Принцип роботи схеми мобільної IP-адреси

Відповідно до рис. 5.9, схема мобільної IP-адреси працює таким чином:

- відправлення пакета хосту на його базову точку;
- повернення поточної адреси;
- передача сигнального пакета за поточною адресою;
- передача подальших пакетів за поточною адресою.

З рис. 5.9 видно також недоліки підходів, що ґрунтуються на базовій точці, для великомасштабних мереж. Для встановлення зв’язку з мобільною сутністю клієнт має спершу зв’язатися з базовою точкою, яка може перебувати зовсім не там, де сама сутність, у результаті чого зростають затримки зв’язку.

Рішення, що застосовується в мобільній телефонії, – це дворівнева схема. Устанавлюючи зв’язок із мобільною сутністю, клієнт спочатку перевіряє локальний реєстр, щоб визначити, чи можна зв’язатися з клієнтом локально.

Якщо не можна, то здійснюється контакт із базовою точкою сутності для визначення її поточного місцезнаходження.

Інший суттєвий недолік підходів, що грунтуються на базовій точці, полягає в її *обов'язковій фіксованості*, тобто завжди має гарантуватися існування базової точки, інакше зв'язатися із сутністю не вдасться. Встановити зв'язок ще складніше, якщо сутність має тривалий строк існування й постійно переміщується у віддаленій від базової точки частини мережі. У цьому разі краще дозволити базі переміщуватися разом із хостом.

Рішення цієї проблеми полягає в тому, щоб *реєструвати базову точку в традиційній службі іменування* й вимагати від клієнта *пошуку спершу базової точки*. Оскільки місцезнаходження базової точки можна вважати порівняно стабільним, після встановлення його можна успішно кешувати.

5.2.4 Ієрархічні підходи

Двоетапний підхід до локалізації сутностей, що ґрунтуються на базовій точці, можна узагальнити для безлічі рівнів. Розглянемо спочатку загальний підхід до ієрархічної схеми локалізації, а потім подамо способи його оптимізації. В основі такого підходу лежить *служба локалізації Globe*, яка є службою локалізації загального призначення, поданою низкою ієрархічних служб локалізації, реалізованих у персональних системах зв'язку (*personal communication systems*).

Основний механізм. В ієрархічній схемі мережу поділяють на **домени**, як у ієрархічній організації DNS. Домен верхнього рівня охоплює мережу цілком, кожний домен поділяють на низку **піддоменів**. Домен найнижчого рівня, названий **листовим доменом** (*leaf domain*), зазвичай відповідає локальній мережі в комп'ютерних мережах або стільнику в мобільній телефонії.

Також, аналогічно DNS й іншим ієрархічним системам іменування, кожний домен *D* має асоційований з ним вузол напрямку *dir(D)*, що відслідковує сутності домена, формуючи **дерево вузлів напрямку.** Вузол напрямку домена верхнього рівня, названий **кореневим вузлом напрямку** (*root directory node*), містить відомості про всі сутності. Таку узагальнену організацію мережі з доменів і напрямних вузлів показано на рис. 5.10.

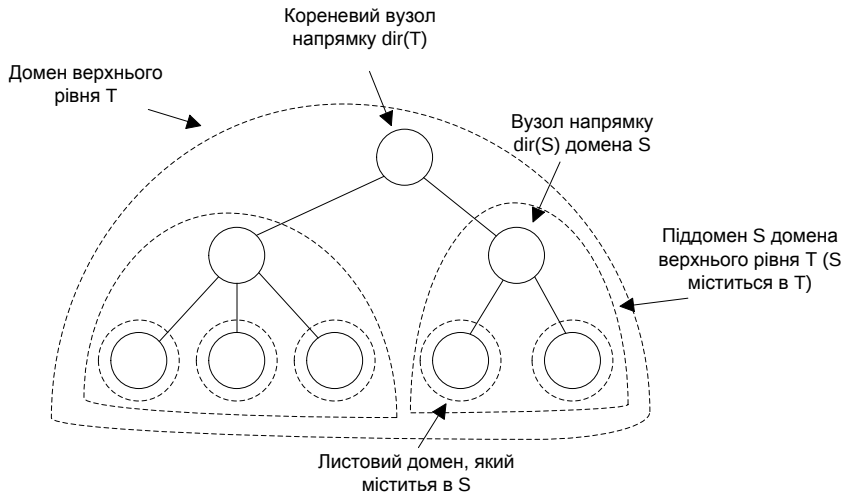


Рис. 5.10. Ієрархічна організація служби локалізації у вигляді доменів, кожний з яких має асоційований вузол напрямку

Щоб відслідковувати місцезнаходження сутностей, кожна сутність, яка перебуває в домені *D*, представлена **локалізуючим записом** (*location record*) у вузлі напрямку *dir(D)*. Локалізуючий запис для сутності *E* у вузлі напрямку *N* листового домена *D* містить поточну адресу сутності в цьому домені, а вузол напрямку *N'* у домені наступного рівня *D'*, у який входить *D,* містить у локалізуючому записі для *E* тільки вказівник на *N*. Так само і батьківський вузол містить у локалізуючому записі для *E* тільки вказівник на *N'*. Відповідно, на кореневому вузлі перебувають локалізуючі записи для всіх сутностей, і кожний такий запис містить вказівник на вузол напрямку в домені нижнього рівня, у якому перебуває сутність, що ідентифікується цим записом.

Сутність може мати кілька адрес, наприклад коли вона реплікована. Якщо сутність має адреси в листових доменах *D1* і *D2* відповідно, то напрямний вузол найменшого з доменів, у який входять *D1* і *D2*, міститиме два вказівники – за адресою на кожний піддомен. Це потребує узагальненої організації мережі (рис. 5.11).

Розглянемо, як у такій ієрархічній службі здійснюється операція пошуку. Як показано на рис. 5.12, клієнт, який бажає знайти сутність *E*, надсилає запит на пошук вузла напрямку листового домена *D,* за яким цей клієнт закріплений. Якщо вузол напрямку не містить локалізуючого запису для цієї сутності, то у цей час вона перебуває за межами домена *D.* Вузол пересилає

запит своєму батьківському вузлу, що є більшим доменом, ніж дочірній. Якщо й батьківський домен не містить локалізуючого запису для *E*, то запит на пошук передається на рівень вище.

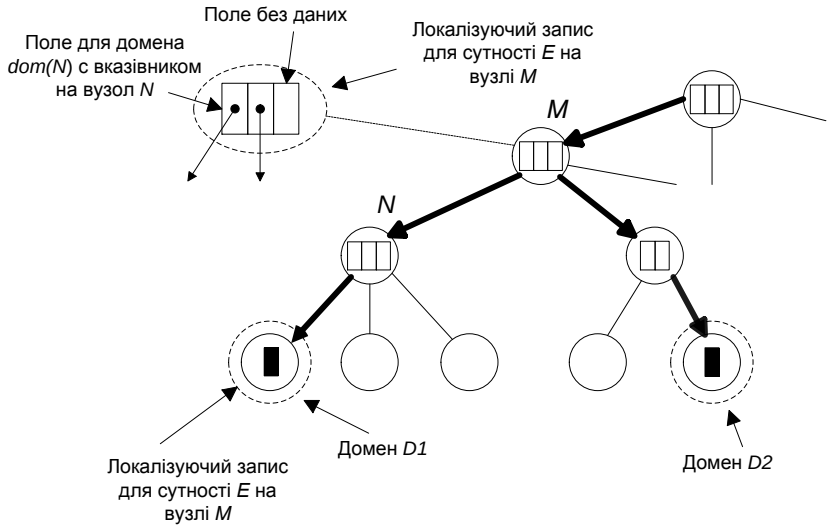


Рис. 5.11. Приклад зберігання інформації про сутність, яка має дві адреси в різних листових доменах

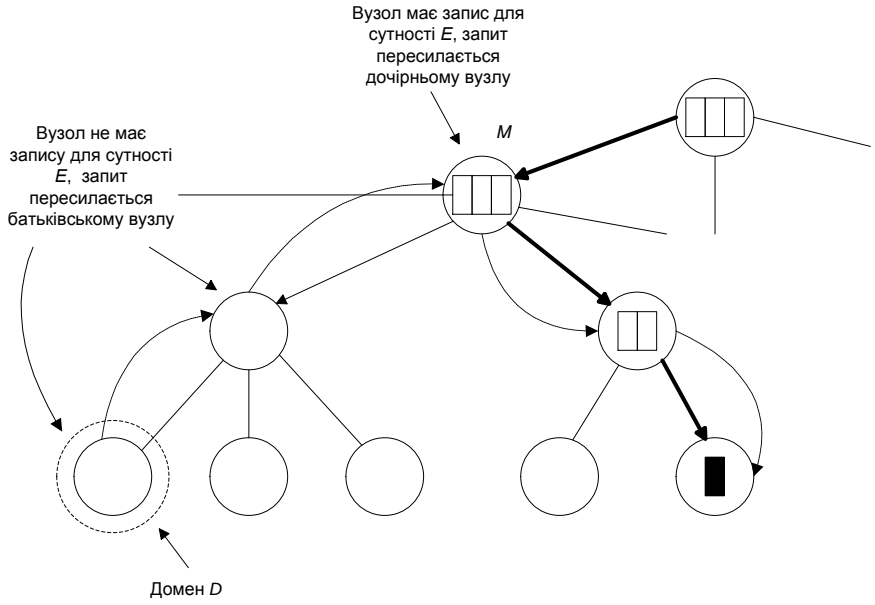


Рис. 5.12. Пошук сутності в ієрархічній службі локалізації

Після того як запит досягне напрямного вузла *M*, де зберігається локалізуючий запис для сутності *E*, визначається, що *E* міститься, наприклад, у домені *dom (M)*, вміст якого відображено на вузлі *M*. На рис. 5.12 показано, що на *M* зберігається локалізуючий запис, що містить вказівник на один з піддо-

менів. Запит на пошук буде пересланий напрямному вузлу цього піддомену, який у свою чергу, перенаправить його далі, вниз по дереву, поки він не досягне листового вузла. Локалізуючий запис, що зберігається на листовому вузлі, містить адресу сутності *E* в цьому листовому домені. Цю адресу і буде повернуто клієнту, який ініціював пошук.

Щодо ієрархічних служб локалізації слід зробити важливе зауваження: операція пошуку розгортається поступово. Сутність шукають у кільці, центром якого є клієнт, що відправив запит, та яке крок за кроком збільшується. Область пошуку розширюється кожного разу, коли запит на пошук пересилається напрямному вузлу верхнього рівня. У найбільш несприятливому випадку пошук продовжується доти, доки запит не досягне кореневого вузла. Оскільки кореневий вузол містить локалізуючі записи всіх сутностей, запит після цього буде пересилатися від вказівника до вказівника, поки не добереться до одного з листових вузлів.

Операції оновлення виконуються аналогічно (рис. 5.13). Розглянемо сутність, для якої в листовому домені *D* було створено репліку. Необхідно вставити в напрямний вузол адресу цієї репліки, починаючи з листового вузла *dir (D)* домену *D*, який пересилає запит на вставку своєму батькові, який також пересилає запит на вставку, і так доти, доки він не досягне напрямного вузла *M*, у якому вже є локалізуючий запис для *E* .

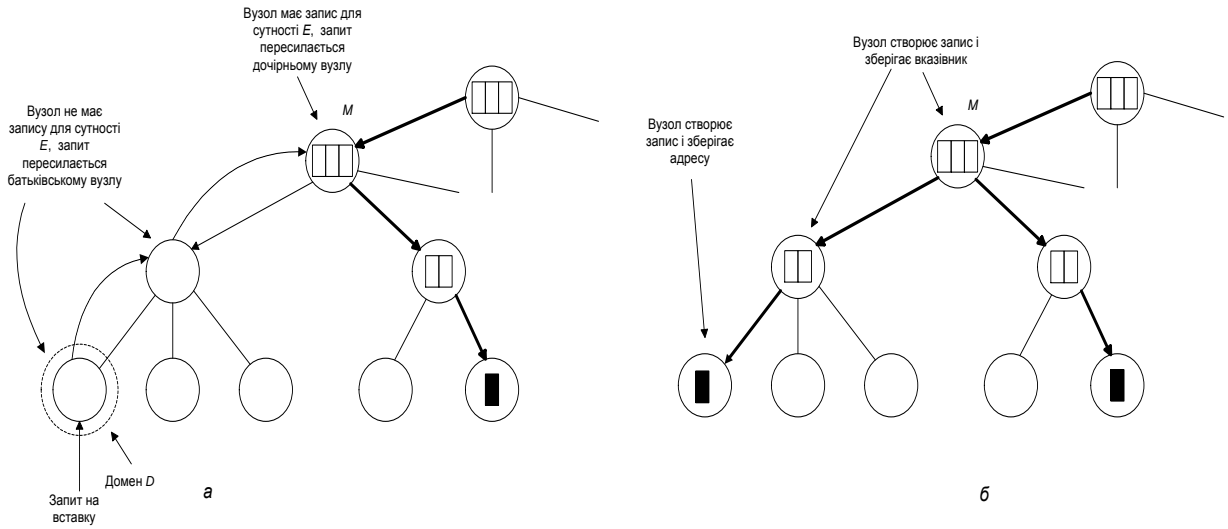


Рис. 5.13. Запит на вставку пересилається першому вузлу, що має інформацію про сутність *E* (*а*); створено ланцюжок передачі вказівників до листового вузла (*б*)

Вузол *M* збереже вказівник у локалізуючому записі для сутності *E*, що посилається на дочірній вузол, від якого надійшов запит на вставку. Після цього дочірній вузол також створює локалізуючий запис для сутності *E*, який містить вказівник на вузол більш низького рівня, від якого запит надійшов до нього. Цей процес триває, поки не досягне листового вузла, який ініціював вставку, а цей вузол створює запис із адресою сутності в асоційованому з ним листовому домені.

Таким чином, вставка адреси зумовлює утворення ланцюжка вказівників, який направлено зверху вниз і починається з направляючого вузла найнижчого серед вузлів, які мають локалізуючий запис для сутності *E* рівня. Альтернативою є створення локалізуючого запису до передачі запиту батьківському вузлу, тобто ланцюжок вказівників формується знизу вгору. Перевага цього варіанта полягає в тому, що адреса для пошукових запитів стає доступною миттєво, тому якщо батьківський вузол тимчасово недоступний, то адресу можна буде знайти в домені, зв'язаному з поточним вузлом.

Операція видалення аналогічна операції вставки. Якщо потрібно видалити адресу сутності *E* з листового домена *D* вузла напрямку – *dir(D),* то надходить запит на видалення цієї адреси з локалізуючого запису для сутності *E*. Якщо цей локалізуючий запис виявляється порожнім, тобто інших адрес для сутності *E* в домені *D* немає, то запис можна видаляти. У цьому разі батьківський вузол для вузла *dir(D)* має видалити свій вказівник на *dir(D)*. Якщо локалізуючий запис для сутності *E* на батьківському вузлі став порожнім, то цей запис також може бути видалений, при цьому потрібно повідомити наступний рівень в ієрархії. Цей процес триває доти, доки, видаливши черговий вказівник із чергового локалізуючого запису, не виявимо там ще один або доки не доберемося до кореневого вузла.

Кешування вказівників. Ієрархічна служба локалізації призначена для підтримки мобільних сутностей, тобто сутностей, місцезнаходження яких постійно змінюється. У традиційних службах іменування відображення імені на адресу передбачається незмінним, як мінімум для вузлів глобального й адміністративного рівнів. Це означає, що збереження результатів пошуку цих вузлів у локальному кеші може істотно підвищити ефективність роботи.

Локальне кешування адрес для служб локалізації зазвичай неефективне, тобто зберігати адресу знайденої мобільної сутності недоцільно у зв'язку з тим, що вона наступного разу може виявитися зовсім в іншому місці. У зв'язку з цим виникає потреба виконати повну процедуру пошуку, що неминуче робить ієрархічні служби локалізації значно дорожчими, ніж більшість служб іменування.

Кешування ефективне лише в тому разі, якщо кешовані дані змінюються рідко. Якщо мобільна сутність *E* регулярно переміщається в межах домена *D*, то *E* постійно змінює поточну адресу, однак її ланцюжок вказівників від кореневого до напрямного вузла *dir(D)* не змінюється. Інакше кажучи, місце, де зберігається найбільш актуальна інформація про місцезнаходження *E*, залишається тією самою, тобто у вузлі напрямку *dir(D)*. Таким чином, ефективність кешування посилань на вузол напрямку є очевидною.

Зазвичай якщо *D* – найменший домен, у межах якого мобільна сутність регулярно переміщується, то слід починати пошук поточного місцезнаходження *E* з вузла напрямку *dir(D)*, а не з якого-небудь іншого вузла. Цей підхід використовується у службі локалізації Globe, називають його **кешуванням вказівників** (pointer cashing). Посилання на *dir(D)* може кешуватися кожним вузлом на шляху до листового вузла, з якого починався пошук, як показано на рис. 5.14.

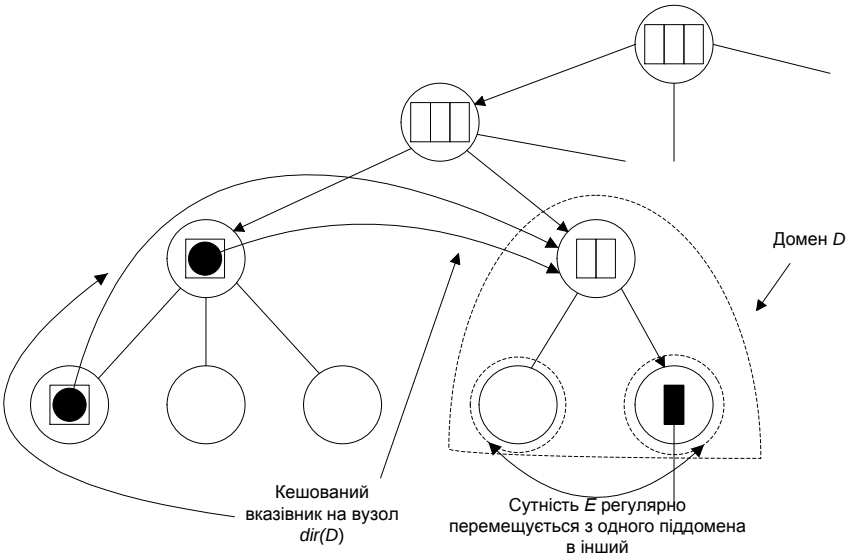


Рис. 5.14. Кешування вказівників на напрямний вузол домена найнижчого рівня з тих, де сутність перебуває більшу частину часу

Подальші вдосконалення цього підходу можливі, якщо не дозволяти вузлу *dir(D)* зберігати вказівник на піддомен, у якому в цей час перебуває *E*, а замінити його поточною адресою *E*. У поєднанні з кешуванням вказівників операція пошуку може бути реалізована лише за два кроки: перший крок вимагає перегляду локального кешу вказівників, які виводять прямо на відповідний вузол напрямку; другий крок виконує запит до вузла з поверненням поточної адреси *E*.

Незважаючи на те, що принцип кешування вказівників у ієрархічних службах локалізації працює, є безліч нюансів, які вимагають особливої уваги. Один з них полягає у виборі вузла напрямку, який найбільш підходить для зберігання поточної адреси мобільної сутності.

Приклад. Припустимо, що користувач з мобільним комп'ютером регулярно переміщується між двома різними містами, наприклад Києвом та Одесою.

Коли користувач перебуває в Києві, можна припустити, що він регулярно змінюватиме своє місцезнаходження всередині міста. У зв'язку з цим варто зберігати його поточне місцезнаходження у вузлі напрямку, що відповідає домену Київ. Така сама модель поведінки буде характерна для користувача й в Одесі.

Однак у разі, якщо користувач весь свій час літає із Києва в Одесу, краще зберігати його поточне місцезнаходження у вузлі напрямку більш високого рівня, наприклад такого, який відповідає Україні, незалежно від того, де перебуває користувач – у Києві чи в Одесі.

У такому разі відкритим залишається запитання, чи відповідає вміст кешу дійсності, тобто чи є адреса правильною. Наприклад, якщо користувач, який перебуває у відрядженні в Харкові, одержав так багато заявок з Києва, що вирішив обробити їх в іншому центральному офісі й почав відправляти всі запити, які надходять на його ім'я в цей район Києва, тобто для служби локалізації в нього з'явилася постійна адреса в домені центрального офіса в Києві. На всі запити на пошук з Києва має повертатися нова адреса, а не кешований вказівник на вузол напрямку в Харкові, як показано на рис. 5.15.

Проблеми масштабування. Одна з основних проблем ієрархічних служб локалізації полягає в тому, що кореневий вузол має зберігати локалізуючі записи й обробляти запити до всіх сутностей. Обсяг збереженої інформації у цьому разі не є головною проблемою. Локалізуючий запис невеликий, він містить лише ідентифікатор сутності й один або більше вказівників на вузли напрямку нижнього рівня. Якщо розмір кожного локалізуючого запису приб-

лизно становить 1 Кбайт, то для зберігання, наприклад, мільярда сутностей необхідний лише 1 терабайт. Цей обсяг забезпечать 10 дисків по 100 Гбайт.

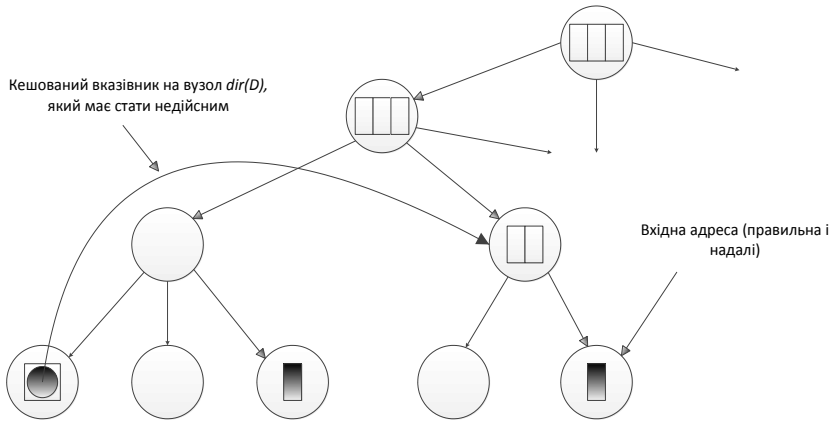


Рис. 5.15. Запис у кеші стає неправильним, оскільки повертається нелокальна адреса, у той час як є локальна

Проблема полягає в тому, що без вживання спеціальних заходів кореневий вузол має обробляти таку кількість запитів на пошук і відновлення, що неминуче виникне вузьке місце. Розв’язати цю проблему можна за рахунок розбивання кореневого вузла й інших вузлів напрямків верхнього рівня на **вкладені вузли**. Кожний вкладений вузол відповідатиме за обробку запитів до певного піднабору сутностей, які підтримуються службою локалізації.

Щоб розв’язати проблему масштабованості, недостатньо просто поділити вузли верхнього рівня на частини. Розглянемо розбиття кореневого вузла на 100 вкладених вузлів. Виникає запитання, де перебуватиме фізично кожний вкладений вузол у мережі, охопленій службою локалізації.

Одна з можливостей вирішити проблему масштабованості – розмістити вкладені вузли відповідно до централізованого підходу недалеко один від одного, наприклад зібрати їх у кластер. Кореневий вузол у цьому разі реалізується за допомогою паралельного комп’ютера типу *COW* або *MPP*. Однак, незважаючи на те, що обчислювальної потужності буде достатньо, пропускної здатності вхідних і вихідних мережних з’єднань для виконання всіх запитів може не вистачити.

Отже, більш вдалою альтернативою є рівномірний розподіл вкладених вузлів мережею. Якщо зробити це неправильно, то виникне проблема масштабування, що призведе до непрацездатності системи в цілому.

Приклад. Розглянемо знову мобільного користувача, який переміщується переважно між Києвом і Одесою. Кореневий вузол поділено на вкладені вузли, і виникає запитання, який вкладений вузол зробити відповідальним за цього користувача.

Припустімо, для постійного зберігання локалізуючих записів цього користувача обрано вкладений вузол, що перебуває у Фінляндії. Незважаючи на проблему кешування вказівників, це означатиме, що запит, наприклад, із Бразилії, перед тим як він буде переданий на вузол напрямку в Одесі, має пройти через кореневий вузол у Фінляндії. Однак, як показано на рис. 5.16, розумніше було б пропустити запит через вкладений вузол, розташований у Одесі.

Рис. 5.16. Ілюстрація проблем масштабування у разі рівномірного розподілу вкладених вузлів розділеного кореневого вузла

Запитання про те, котрому вкладеному вузлу за яку сутність відповідати, для великомасштабних служб локалізації залишається відкритим. Можна розв’язати цю проблему, якщо використати те місцезнаходження, у якому створено сутність *E*. Зокрема, вкладений вузол кореневого вузла, розташований неподалік від місця, де створено сутність *E*, може бути призначений відповідальним за обробку запитів кореневого рівня, які стосуються *E*. Це рішення є ефективним для сутностей, які мають тенденцію перебувати

неподалік від місця, де їх створено, але якщо сутність переміщується у віддалене місце, проблема залишається.

5.3 Видалення сутностей, на які немає посилань

Служби іменування й локалізації надають **глобальні служби посилань** на сутності. Доки у службі є посилання на сутність, доти ця сутність є доступною користувачам і може використовуватися. Після того як доступ до сутності припиняється, її варто **видалити.** У багатьох системах видалення сутностей реалізується явно.

Приклад. Якщо процес *P* знає, що він – останній з процесів, які використовуватимуть деякий файл, причому в майбутньому цей файл жодному процесу не знадобиться, *P* може під час свого завершення видалити цей файл.

Керування видаленням сутностей у розподілених системах є достатньо складним. Так, зокрема, часто невідомо, де в системі створено посилання на сутність, яка видаляється, з наміром скористатися ними пізніше для доступу до цієї сутності. Якщо пізніше до видаленої сутності дійсно відбудеться спроба доступу за допомогою одного з таких посилань, то це призведе до помилки.

Натомість, неприпустимо також взагалі не видаляти сутності через брак гарантій, що на неї немає посилань. Якщо посилань все-таки немає, то сутність, яка продовжує споживати ресурси, ніколи не використовуватиметься. Зрозуміло, що ця сутність – просто **сміття** … її варто видалити.

Розподілені системи надають механізми автоматичного видалення сутностей, на які немає посилань, для зниження впливу на ефективність їх функціонування проблем, пов’язаних із видаленням сутностей. Ці механізми відомі під загальною назвою **«розподілені збирачі сміття»** (distributed garbage collectors). Розглянемо взаємини сутностей іменування й локалізації з автоматичним збиранням «нічийних» сутностей (сутностей, які нікому не належать).

5.3.1 Об'єкти, на які немає посилань

Щоб зрозуміти, як відбувається збирання сміття, розглянемо збирання розподілених об'єктів, зокрема видалених. Нагадаємо, що **видалений об'єкт** реалізується так, що його стан цілком перебуває на сервері об'єктів, а клієнт працює винятково із замісником. Посилання на видалений об'єкт зазвичай реалізуються парою (замісник, скелетон), у якій замісник клієнта містить усю інформацію, необхідну для роботи з об'єктом через зв'язаний з ним скелетон, реалізований на сервері, а скелетон разом із замісником бере участь в адмініструванні, потрібному для збирання сміття, тобто процедури збирання сміття, сховані як від користувача, так і від самого об'єкта. Відзначимо, що й об'єкт може містити видалені посилання на інші об'єкти, наприклад локальні посилання на замісників об'єктів, які видалено. Аналогічно видалені посилання можуть передаватися іншим процесам за рахунок копіювання в них замісника, зв'язаного з процесом.

Отже, доступ до об'єкта можна отримати лише за умови, що є віддалені посилання на нього, а об'єкт, на який немає віддалених посилань, можна видаляти із системи. Проте наявність віддалених посилань не означає, що об'єкт буде доступним завжди, оскільки з різних причин можливе виникнення двох об'єктів, які посилаються один на одного і не мають інших посилань. Ця ситуація узагальнюється на ланцюжок об'єктів, кожний з яких посилається на сусідні об'єкти, які також варто виявляти й видаляти.

Модель посилань на об'єкти можна подати у вигляді графа, кожний вузол якого – це об'єкт. Ребро, яке прямує від вузла *M* до вузла *N*, відображає той факт, що об'єкт *M* містить посилання на об'єкт *N*. Підмножина об'єктів, у яких немає посилань один на одного, відома під назвою **кореневого набору** (root set). Об'єкти кореневого набору зазвичай представляють системні служби, користувачів та ін.

Граф посилань показано на рис. 5.17, на якому всі білі вузли зображують об'єкти, на які в об'єктів кореневого набору немає прямих або непрямих посилань, тому вони можуть бути видаленими.

В однопроцесорних системах виявлення й видалення об'єктів, на які немає посилань, нескладне порівняно з тим, як це відбувається в розподіле-

них системах. Оскільки об’єкти в цьому разі розкидані по безлічі машин, розподілене збирання сміття потребує мережного зв’язку, який істотно впливає на продуктивність і масштабованість рішень. Крім того, як зв’язок, так і машини та процеси, піддані збоям, які здатні ще більше ускладнити процеси збирання сміття.

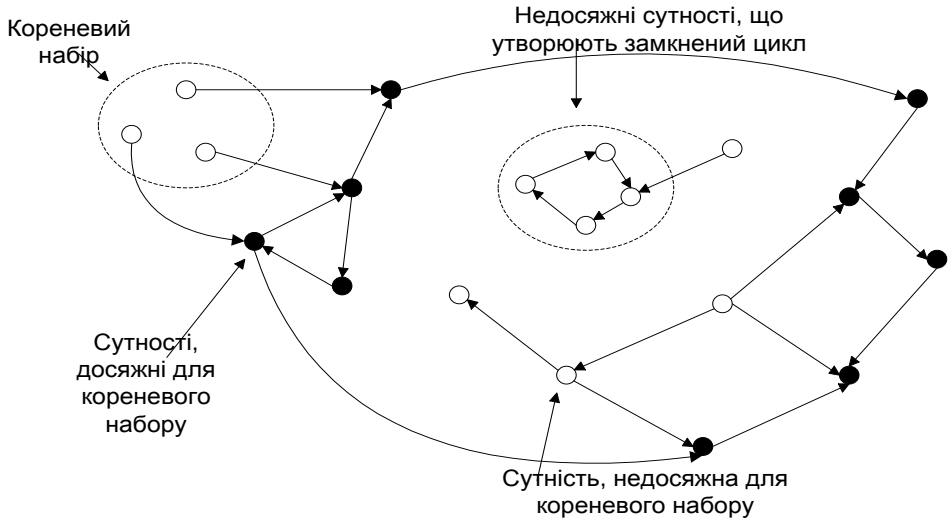


Рис. 5.17. Приклад графу об’єктів, що містять посилання один на одного

5.3.2 Підрахунок посилань

Метод перевірки можливості видалення певного об’єкта, поширеного в однопроцесорних системах, полягає в тому, щоб підраховувати посилання на цей об’єкт, тобто під час створення кожного посилання на об’єкт лічильник посилань на цей об’єкт збільшується на одиницю, а коли посилання на об’єкт знищуються, значення лічильника зменшується. Як тільки значення лічильника дорівнюватиме нулю, об’єкт можна видаляти.

Простий підрахунок посилань. Простий підрахунок посилань у розподілених системах зумовлює часткове виникнення проблем через ненадійність зв’язку. Припускають, що об’єкт зберігає свій лічильник посилань у відповідному скелетоні, яким керує сервер об’єкта, відповідальний за цей об’єкт, що показано на рис. 5.18.

Коли процес *P* створює посилання на видалений об’єкт *O*, він, як показано на рис. 5.18, установлює замісника цього об’єкта у свій адресний простір.

Для збільшення лічильника посилань замісник надсилає повідомлення *(1)* скелетону об’єкта й очікує, коли він поверне підтвердження. Однак, якщо підтвердження загубиться *(2)*, то замісник повторно надішле повідомлення *(3)*. Якщо не вживати ніяких заходів для виявлення повторних повідомлень, то скелетон може помилково збільшити лічильник ще раз. На практиці порівняно просто виявляти повторні повідомлення.

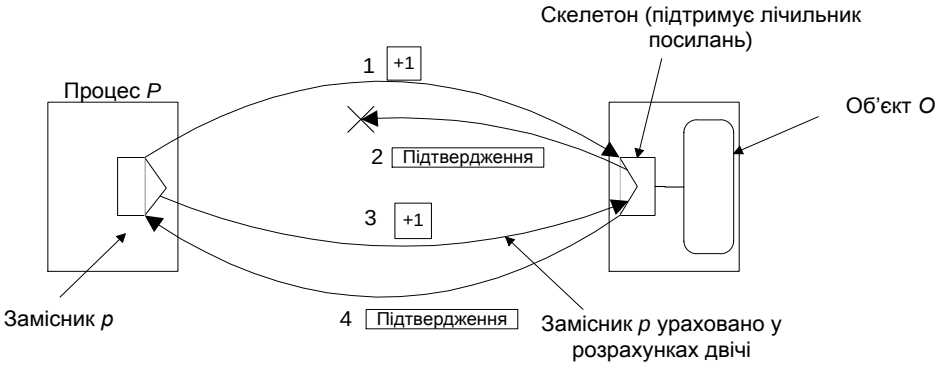


Рис. 5.18. Проблема збереження правильності лічильника посилань в умовах ненадійного зв’язку

Аналогічні проблеми можуть виникнути й у разі знищення віддаленого посилання, коли замісник надсилає повідомлення, яке зменшує лічильник посилань. Якщо підтвердження знову втратиться, то повторне надсилання повідомлення призведе до повторного, цього разу помилкового, зменшення лічильника. Таким чином, під час розподіленого підрахунку посилань важливо виявляти повторні повідомлення та ігнорувати їх.

Ще одна проблема, яку необхідно розв’язати, виникає у процесі копіювання віддаленого посилання в інший процес, тобто якщо процес *P1* передає посилання процесу *P2*, то об’єкт *O*, точніше його скелетон, може не отримати інформації щодо створення нового посилання. Таким чином, якщо процес *P1* вирішить знищити своє посилання, то вміст лічильника посилань може дорівнятися до нуля та є ймовірність для об’єкта *O* бути видаленим до того, як процес *P2* зможе з ним зв’язатися. Цю проблему ілюструє рис. 5.19, *а*.

Розв’язання цієї проблеми полягає в тому, щоб вимагати від *P1* повідомляти скелетон об’єкта про те, що він передаватиме посилання процесу *P2*, який не видалятиме своє посилання доти, доки скелетон не підтвердить йому, що знає про створення посилання, що продемонстровано на рис. 5.19, *б*. Підтвер-

дження, яке надсилається від об'єкта *O* процесу *P2*, повідомляє його, що об'єкт *O* зареєстрував посилання, яке дозволяє *P1* пізніше видалити своє посилання. Доки процес *P2* не впевнений, що об'єкт *O* знає про наявність у нього посилання, *P1* не може вимагати від об'єкта *O* зменшення лічильника посилань.

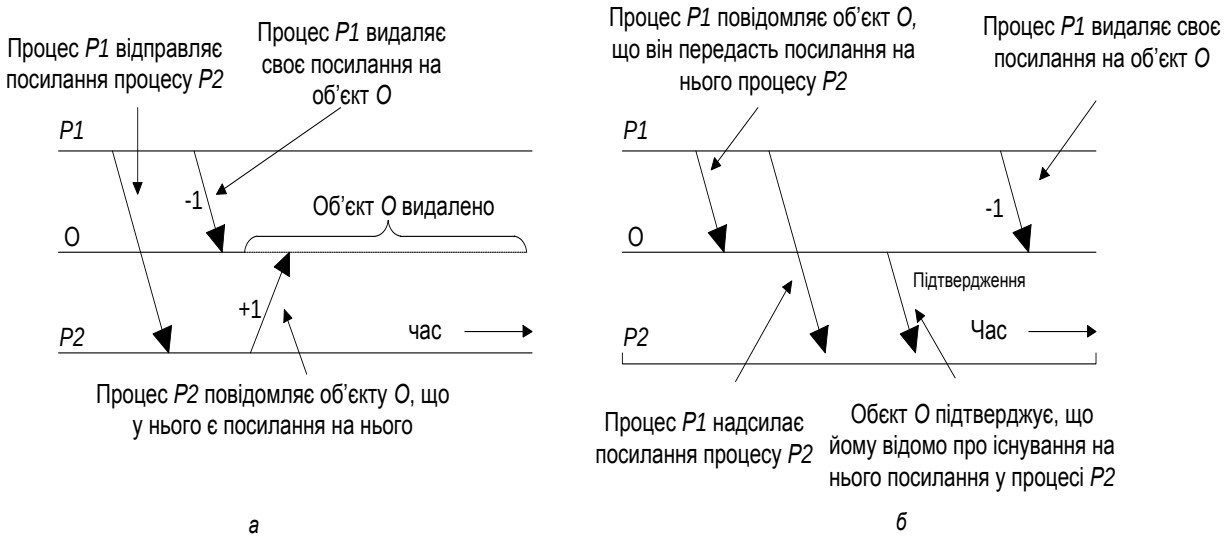


Рис. 5.19. Копіювання посилання в інший процес: *а* – запізнене збільшення лічильника; *б* – розвязання проблеми

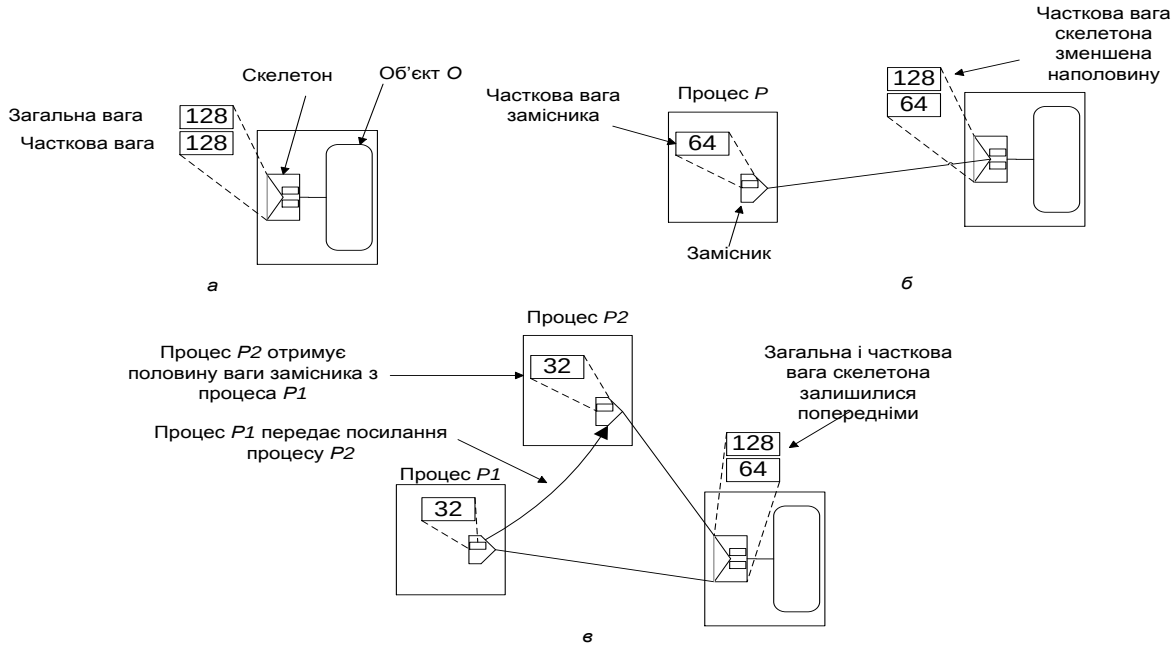


Рис. 5.20. Вихідне співвідношення значень лічильника посилань: *а* – у разі зваженого підрахунку посилань; *б* – присвоєння значення лічильника посилань під час створення нового посилання; *в* – присвоєння значення лічильника посилань під час копіювання посилання

Крім надійного зв’язку, передача посилання вимагає трьох повідомлень, що у великомасштабних розподілених системах зменшує продуктивність.

Удосконалені механізми підрахунку посилань. Простий розподілений підрахунок посилань вимагає дотримання певних умов у період між збільшенням і зменшенням лічильника посилань. Ці умови можна не виконувати, якщо відбувається лише зменшення лічильника. Таке рішення реалізовано **у зваженому підрахунку посилань** (*weighted reference counting*), під час якого кожний об’єкт має **фіксовану загальну вагу**. Під час створення об’єкта ця вага зберігається в асоційованому з ним скелетоні разом з його **частковою вагою**, яка ініціалізується загальною вагою, як показано на рис. 5.20, *а*.

У разі створення нового віддаленого посилання половина часткової ваги, що зберігається в скелетоні об’єкта, присвоюється новому замісникові, як показано на рис. 5.20, *б*, половина, що залишилася, зберігається у скелетоні. Коли видалене посилання дублюється, наприклад, під час його передачі з процесу *P1* у процес *P2*, половина часткової ваги замісника з *P1* передається замісникові процесу *P2*, у який відбувається копіювання, а друга половина залишається у замісника процесу *P1*, як показано на рис. 5.20, *в*.

У разі знищення посилання скелетон об’єкта надсилає повідомлення, яке зменшує лічильник, потім від загальної ваги віднімається часткова вага знищеного посилання. Коли загальна вага дорівнюватиме нулю, об’єкт можна видаляти, передбачаючи, що повідомлення не губляться і не дублюються.

Основним недоліком зваженого підрахунку посилань є те, що кількість створюваних посилань є обмеженою, тобто коли часткова вага скелетона і віддалених посилань досягне нуля, створювати або копіювати нові посилання буде неможливо. Розв’язати цю проблему можна за допомогою **непрямих посилань.**

Припустімо, процес *P1* хоче передати посилання процесу *P2*, але часткова вага його замісника дорівнює одиниці, як показано на рис. 5.21, тоді *P1* створює у своєму адресному просторі скелетон *s′* з потрібною загальною вагою й частковою вагою, яка дорівнює загальній. Такий принцип є аналогічним створенню в адресному просторі об'єкта скелетона *S*. Потім процесу *P2* пересилається замісник, якому передається половина часткової ваги скелетона *s′* , а друга половина ваги залишається в *s′* для інших замісників.

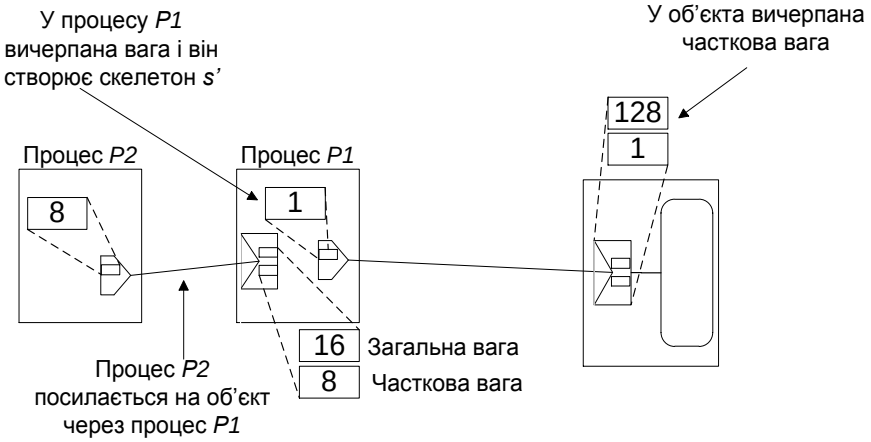


Рис. 5.21. Створення непрямого посилання у разі вичерпання часткової ваги

Якщо загальна вага скелетона s' дорівнюватиме одиниці ($s'=1$), то у такому разі підхід з використанням непрямих посилань стане подібним до передачі вказівника від процесу *P1* до процесу *P2*. Якщо процес *P2*, у свою чергу, захоче передати своє посилання, він може створити ще один вказівник, який передаватиметься. Найбільш суттєва проблема передачі вказівників полягає в тому, що довгі ланцюжки значно знижують продуктивність і достатньо вразливі у разі збоїв.

Альтернативою непрямим посиланням може бути **підрахунок поколінь посилань** (*generation reference counting*). Припустімо, кожне видалене посилання являє собою пару (замісник, скелетон), причому скелетон належить тому самому адресному простору, що й об'єкт. Кожний замісник зберігає **лічильник копій** і **номер покоління.** Для створення нового посилання (замісник, скелетон) номер покоління відповідного замісника встановлюється в нуль. Оскільки поки ще не було зроблено копій замісника, його лічильник копій також дорівнює нулю.

Копіювання віддаленого посилання (замісник, скелетон) в інший процес виконується за допомогою пересилання йому копії замісника. У цьому разі лічильник копій вихідного замісника збільшується на одиницю, а лічильник копій скопійованого виставляється в нуль. Однак оскільки цей замісник скопійований, він належить до наступного покоління, з огляду на це його номер покоління на одиницю більший, ніж у вихідного замісника, що й демонструє рис. 5.22.

Скелетон підтримує таблицю *G*, у якій *G[i]* означає кількість створених копій для покоління *L*. Якщо замісник *p* видаляється, то повідомлення про видалення надсилається скелетону, який містить номер покоління замісника, наприклад *k*, і кількість копій, утворених від замісника *p*, наприклад *n*. Скелетон приводить *G* у відповідність до дійсності, спочатку зменшуючи на одиницю *G[k]* і показуючи таким чином, що видалено посилання *k*-го покоління. Потім він збільшує *G[k +1]* на *n*, щоб показати, що видалене посилання створило *n* нащадків, тобто було скопійовано в *n* посилань наступного покоління (відзначимо, що скелетон має спочатку створити елемент *G[k +1]*, якщо раніше він нічого не знав про покоління (*k +1*)). Якщо всі елементи *G[i]* дорівнюють нулю, то на об'єкт більше немає посилань і його можна видалити.

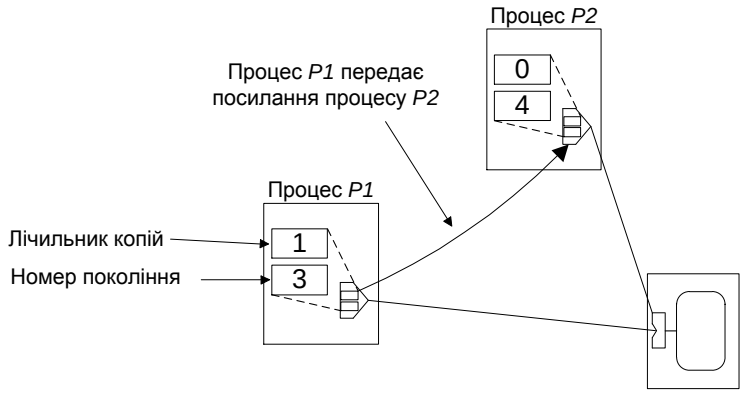


Рис. 5.22. Створення й копіювання видалених посилань у процесі підрахунку поколінь посилань

Підрахунок поколінь посилань також вимагає надійних комунікацій, але здатний забезпечувати копіювання посилань, не звертаючись для створення копій до скелетона.

5.3.3 Організація списку посилань

Інший підхід до керування посиланнями полягає в тому, щоб змусити скелетон відслідковувати замісників, які містять посилання на нього, тобто замість підрахунку посилань скелетон має підтримувати повний список усіх замісників, які на нього вказують. Такий **список посилань** (*reference list*) має такі важливі властивості: внесення замісника до списку посилань не дає ні-

яких результатів, якщо він у списку вже є; спроба видалення замісника, якого немає у списку, не призводить до виконання ніяких операцій.

Додавання й видалення замісників у такий спосіб називають **ідемпотентними** *(idempotent)* **операціями**, які характеризуються тим, що їх багаторазове виконання не впливає на кінцевий результат, зокрема, під час створення нового посилання на об'єкт-процес, що створив посилання, об'єкт-процес може неодноразово відправляти повідомлення скелетону об'єкта, вимагаючи внести посилання до списку посилань. Він припиняє надсилати повідомлення після одержання підтвердження про доставку. Про видалення посилання можна повідомляти надсиланням (можливо, багаторазово) скелетону повідомлення з вимогою вилучити посилання зі списку. Відзначимо, що операції збільшення і зменшення лічильника не є ідемпотентними операціями.

Таким чином, список посилань не вимагає від зв'язку надійності, механізмів виявлення й відкидання повторних повідомлень, однак необхідно, щоб додавання посилань у список і видалення їх зі списку підтверджувалося. Це суттєва перевага порівняно з механізмами підрахунку посилань.

Списки посилань, які використовуються в *RMI* (*Remote Method Invocation*) мови *Java*, в основу яких покладено метод, відповідно до якого, коли процес *P* створює віддалене посилання на об'єкт, він надсилає свій ідентифікатор скелетону об'єкта, який додає *P* у список посилань. Коли надходить підтвердження, процес створює замісника об'єкта у своєму адресному просторі.

Аналогічно передається посилання іншому процесу, тобто надсилається копія замісника. Коли процес *P1* надсилає копію свого замісника об'єкта *O* процесу *P2*, то процес *P2* спочатку вимагає у скелетона об'єкта додати *P2* у список посилань. Потім процес *P2* вбудовує замісника у свій адресний простір. Проблеми можуть виникнути в тому разі, якщо процес *P1* видалить свого замісника до того, як *P2* подасть запит на вставку список посилань. Якщо запит на видалення, який *P1* надсилає скелетону об'єкта, буде оброблений раніше від запиту на вставку від *P2*, то список посилань може виявитися порожнім, тобто скелетон може дійти неправильного висновку про можливість видалення об'єкта. Це аналогічно підходу

з підрахунком посилань, який наведено на рис. 5.19, *а*, і може бути вирішено у той самий спосіб.

Важливою перевагою використання списку посилань є простота збереження його несуперечності у разі збоїв у процесах. Скелетон об'єкта регулярно перевіряє, чи *всі наявні у списку процеси працездатні*. Це робиться за допомогою надсилання повідомлення *ping*, запитуючи відповідь від процесів, яка визначає їх працездатність, і чи зберігають вони посилання на об'єкт. Процес має негайно відповісти на це повідомлення, оскільки, якщо відповіді не отримано після декількох спроб, скелетон видаляє процес зі списку.

Основний недолік списку посилань полягає в тому, що він погано масштабується, особливо якщо скелетон має відслідковувати багато посилань. Один зі способів збереження керованості полягає в тому, щоб надати можливість скелетону реєструвати посилання на обмежений строк. Якщо замісник до закінчення цього строку не підтверджує скелетону свою реєстрацію, посилання просто видаляється зі списку. Цей підхід називають **орендою** (*lease*).

5.3.4 Ідентифікація сутностей, на які немає посилань

Як зображено на рис. 5.22, набір сутностей у розподілених системах може містити сутності з посиланнями тільки один на одного, тобто недоступні з кореневого набору. Такі сутності мають видалятися, але, на жаль, методи збирання сміття не спроможні виявити подібні сутності.

Слід було розробити метод, який дозволив би відстежити всі сутності в розподіленій системі. Цей метод полягає в перевірці доступності сутностей з кореневого набору з подальшим видаленням усіх недоступних. Його названо **трасувальним збиранням сміття** (*tracing-based garbage collection*). На відміну від розподілених посилань, трасувальному збиранню сміття властиві проблеми *масштабності*, оскільки необхідно відслідковувати всі сутності розподіленої системи.

Примітивне трасування в розподілених системах. Щоб зрозуміти сутність розподіленого трасувального складання сміття, варто розглянути трасування в однопроцесорних системах, найбільш простий підхід до якого використано в алгоритмах збирачів типу «позначай і підмітай», які працюють у два етапи.

У процесі етапу позначення сутності відслідковують за наявними ланцюжками посилань сутності, які виходять із сутностей кореневого набору. Кожна сутність, що може бути досягнута з кореневого набору, позначається за допомогою запису сутності в окрему таблицю. Етап підмітання полягає у ретельній перевірці пам'яті для локалізації «нічийних» сутностей, що вважаються сміттям, яке слід видалити.

Інший варіант роботи збирачів «позначай і підмітай» – триколірне позначення сутностей. Спершу кожну сутність, яку варто перевірити, пофарбовано в білий колір. До кінця етапу позначення всі сутності, доступні з кореня, позначаються чорним, а недоступні залишаються білими. Сірий колір використовується для індикації перебігу етапу позначення, тобто сутність позначається сірим, якщо вона доступна, але посилання, які вона містить, ще потрібно перевірити. Коли всі сутності, на які посилається така сутність, фарбуються сірим, сама вона стає чорною.

Розподілений варіант схеми «позначай і підмітай» реалізовано у системі *Emerald*, у якій кожний процес запускає локальний збирач сміття, причому всі збирачі працюють паралельно, зокрема фарбують замісників, скелетони й самі об'єкти. Спочатку всі вони позначаються білим кольором. Коли об'єкт, розташований в адресному просторі процесу *P*, доступний з кореневого набору, об'єкт позначається сірим, так само як і всі замісники, які містяться в цьому об'єкті. Позначка замісника сірим кольором свідчить про те, що його локальний збирач сміття вказує на необхідність перевірки посилань цього віддаленого об'єкта асоційованим з ним локальним збирачем сміття.

Коли замісник позначається сірим, асоційованому із замісником скелетону надсилається повідомлення, яке позначає сірим і його. Об'єкт, з яким пов'язаний скелетон, позначається сірим слідом за своїм скелетоном, а потім рекурсивно й усі замісники в цьому об'єкті. У цей момент скелетон і асоційований з ним об'єкт позначаються чорним, та відповідне повідомлення надсилається всім асоційованим з ним замісникам. Відзначимо, що хоча в цьому підході скелетону відомі пов'язані з ним замісники, це не означає, що вони досяжні зі скелетона. Пари (*замісник*, *скелетон*) – це винятково *однонапрямний зв'язок* від замісника до скелетона.

Коли замісник одержує повідомлення про те, що асоційований з ним скелетон став чорним, він теж позначається чорним. Інакше кажучи, локальний збирач сміття тепер знає, що віддалений об'єкт, на який іде посилання через замісника, визнано доступним.

Коли всі локальні збирачі сміття завершують етап позначення, тобто коли всі об'єкти, скелетони й замісники мають або білий, або чорний колір, кожний з них окремо збирає всі білі об'єкти, вважаючи їх сміттям. Видалення білих об'єктів означає і видалення асоційованих із ними скелетонів та замісників цих об'єктів.

Основний недолік алгоритму **«позначай і підмітай»** полягає в тому, що граф доступності протягом обох етапів має залишатися незмінним. Інакше кажучи, виконання програми, для якої спочатку було створено процес, буде тимчасово припинено й система розпочне збирання сміття. Для розподілених систем це означає, що всі процеси мають бути синхронізованими, тобто кожен з них має перемкнутися на збирання сміття, після чого всі вони зможуть продовжити свою роботу.

Такий сценарій названо **синхронізацією «усе завмри!»**, зазвичай недоступний для розподілених збирачів сміття, щоб удосконалити які, слід використовувати *інкрементні збирачі сміття*, які дозволяють програмі працювати, періодично виконуючи збирання сміття. На жаль, такі збирачі сміття в розподілених системах погано масштабуються, оскільки працюють паралельно з програмою, яка змінює граф доступності, об'єкти часто позначаються сірим, що зумовить поширення сірих позначок у віддалених процесах. Результатом буде великий трафік повідомлень, здатний знизити загальну продуктивність системи.

Трасування у групах. Розглядаючи проблеми масштабованості, властиві багатьом трасувальним системам збирання сміття, розроблено спосіб, під час якого процеси, які утримують об'єкти, у великих розподілених системах збираються у групи. Збирання сміття виконується всередині груп за рахунок поєднання методів «позначай і підмітай» та підрахунку посилань. Розглянемо базовий алгоритм збирання сміття у групі процесів.

Групу – набір процесів – використовують для подолання проблем масштабованості. Спочатку збирається все сміття у групі, включаючи ланцюжки

посилань, які повністю перебувають усередині групи. Потім формується розширена група більшого розміру, яка містить кілька підгруп, кожна з яких очищена збирачем сміття.

Для реалізації трасування у групі віддалені посилання також реалізовані у вигляді пар (замісник, скелетон). Для кожного об'єкта наявний лише один скелетон, який перебуває в адресному просторі об'єкта, і безліч замісників, здатних зв'язуватися зі скелетоном. Скелетон підтримує лічильник посилань, який підраховує кількість асоційованих із цим скелетоном замісників. Процес може мати максимум одного замісника на кожний розподілений об'єкт.

Як тільки буде сформована група процесів, розпочинає роботу базовий алгоритм збирання сміття, який містить такі п'ять кроків:

– первинне маркування, у процесі виконання якого позначаються тільки скелетони;

– поширення маркування всередині процесів зі скелетонів на замісників;

– поширення маркування між процесами із замісників на скелетони;

– стабілізація за рахунок багаторазового повторення двох попередніх кроків;

– видалення сміття.

Розглянемо, що означає термін «маркування сутності». Алгоритм стосується переважно маркування замісників і скелетонів, проте слід урахувати, що ні замісники, ні скелетони не стосуються кореневого набору сутностей.

Скелетони можуть позначатися як «*нетверді*» або «*тверді*», а замісники – як «*відсутні*», «*нетверді*» або «*тверді*». Якщо скелетон позначений як «твердий», то він досяжний або із замісника процесу, який не входить у групу, або з кореневого об'єкта, який входить у групу, тобто об'єкта, який міститься в кореневому наборі й входить у групу процесу. Скелетон, позначений як «нетвердий», доступний тільки із замісника всередині групи.

Замісник, позначений як «твердий», досяжний з об'єктів кореневого набору, а якщо замісник позначений як «нетвердий», то він досяжний зі скелетона, також позначеного як «нетвердий». Такі скелетони потенційно можуть розміщуватися ланцюжком, до якого не буде доступу в об'єктів кореневого

набору. Замісник, позначений як «відсутній», недоступний ні зі скелетонів, ні з об’єктів кореневого набору. Тільки замісники, позначені як «відсутні», можуть змінювати маркування на «тверді». Замісники, позначені як «нетверді», зберігають це маркування й у подальшому.

Перший крок базового алгоритму збирання сміття полягає у позначенні одних скелетонів як «твердих» або «нетвердих», залежно від того, чи доступний скелетон для замісників, які не входять у групу, що можна легко перевірити за лічильником посилань скелетона. Значення лічильника показує, скільки замісників інших процесів на нього посилаються, причому деякі із цих процесів входять у групу, інші не входять. Якщо серед них є замісники процесів, що не входять у групу, скелетон позначається як «твердий». Підрахувавши кількість замісників, які асоційовані зі скелетоном і входять у групу, й віднявши цю кількість від значення лічильника посилань, можна зрозуміти, чи є у скелетона асоційовані з ним замісники поза групою. Це виконується таким чином: для кожного замісника всередині групи зменшується лічильник посилань відповідного скелетона на одиницю у тому разі, якщо скелетон також перебуває всередині групи. Скелетон усередині групи позначається як «нетвердий», якщо його лічильник посилань має значення «нуль», інакше він доступний поза групою й позначається як «твердий».

Приклад. Перший крок алгоритму ілюструє рис. 5.23, *а*, на якому всі скелетони, крім одного, є «нетвердими». Тільки один скелетон, що позначено як «твердий», доступний замісникові поза групою.

Другий крок алгоритму містить перемикання кожного з процесів на власний збирач сміття, що працює незалежно від глобального збирача сміття й головною функцією якого є поширення маркування зі скелетонів на замісників. Усередині одного процесу замісники досяжні зі скелетона, враховуючи, що замісник і скелетон належать різним об’єктам. У результаті локального поширення позначок замісники будуть позначені так само, як і скелетон. Крім того, якщо замісник буде доступний об’єкту кореневого набору, то його буде позначено як «твердий».

Локальне поширення всередині процесу *P* може відбуватися в такий спосіб. Спочатку всі замісники позначаються як «відсутні». Локальний збирач сміття починає трасування з набору, до якого входять скелетони, що раніше були позначені як «тверді», а також об’єкти кореневого набору. Позначки «тверді» поширюються на всі об’єкти, тобто на локальні об’єкти й замісників, які доступні з цього набору процесів. Другий прохід здійсню-

ється, починаючи зі скелетонів, позначених як «нетверді». Якщо замісники, доступні із цього набору процесів, позначені як «відсутні», їх маркування змінюється на «нетверді»; якщо досяжні замісники вже позначені як «тверді», то за ними зберігається це маркування. Таким чином, після локального поширення кожний замісник процесу одержує одне з трьох маркувань – «відсутній», «нетвердий» або «твердий». Стан, який утворився після локального поширення маркувань у групі процесів, зображено на рис. 5.23, *б*, а початковий стан – на рис. 5.23, *а*.

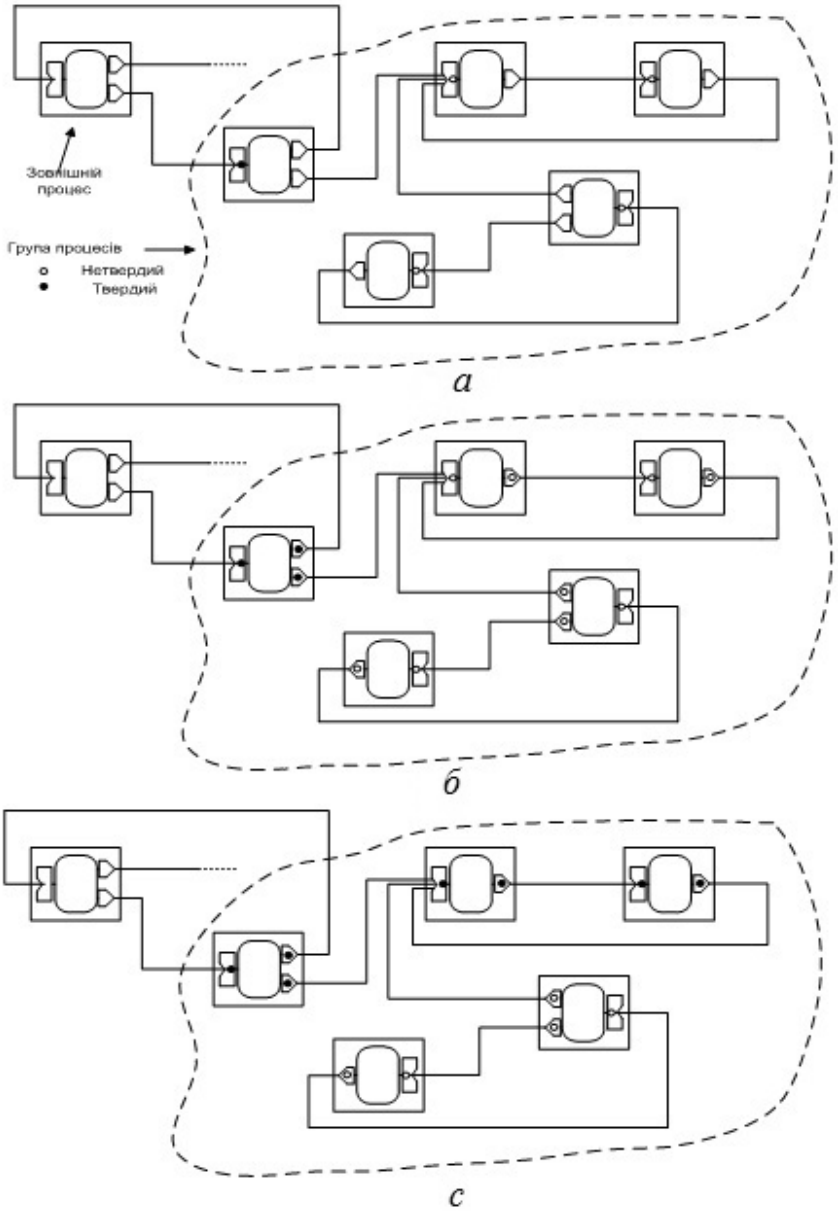


Рис. 5.23. Маркування об’єктів: *а* – початкове маркування скелетона;
б – стан після локального поширення маркування в кожному процесі;
в – підсумкове маркування

Третій крок полягає у поширенні позначок від замісників до асоційованого з ними скелетона, тобто від процесу до процесу. Так, якщо замісник був позначений як «твердий», то асоційованому з ним скелетону надсилається повідомлення про те, що він також позначається як «твердий», оскільки цей скелетон раніше був маркований інакше. Повідомлення надсилається лише тим скелетонам, які входять у групу процесів. Позначки «нетвердий» поширювати не потрібно, оскільки у вихідній розмітці вже враховано, що всі скелетони групи мають позначку «твердий» або «нетвердий».

Четвертий крок зумовлює глобальне поширення позначок «твердий» попереднього кроку, при цьому скелетон, наприклад процесу *P*, може змінити свою позначку з «нетвердий» на «твердий». Ці зміни будуть зумовлені тим, що до скелетона одержать доступ об'єкти, які містяться в кореневому наборі віддаленого процесу. Відповідно, позначка «твердий» спочатку поширюється локально – на замісників процесу *P*, а потім глобально – на суміжні процеси. Інакше кажучи, другий і третій кроки повторюватимуться доти, доки позначки не припинять поширюватися глобально або локально. Коли ситуація стабілізується, тобто припиняться зміни позначок у процесах, що входять у групу, алгоритм перейде до наступного кроку. У прикладі, зображеному на рис. 5.23, результат повторень кроків 2 і 3 приведе до підсумкового позначення, показаного на рис. 5.23, *в*.

П'ятий крок полягає у видаленні недоступних об'єктів, включаючи замісників, а також замісників і скелетонів, позначених як «нетверді». Важливо відзначити, що останні недоступні поза групою або з об'єктів кореневого набору. Інакше кажучи, «нетверді» замісники і скелетони посилаються тільки один на одного й можуть бути видалені.

Збирання сміття може в дійсності виконуватися у вигляді вторинного ефекту локального поширення. Замість явного видалення сутностей на останньому кроці скелетони, позначені як «нетверді», можуть замінюватися посиланням на нуль. Таким чином, вони можуть бути оброблені пізніше, під час повторного запуску локальних збирачів сміття. Крім того, якщо об'єкт, асоційований з таким скелетоном, стає недоступним, то він також може бути оброблений під час повторного проходження. Якщо замісники, які локально посилаються на подібний об'єкт, також стають недоступними, то їх можна позначати як «відсутні» й обробляти так само. З огляду на це для безпечного використання локальних збирачів сміття варто обробляти замісників, позначених як «відсутні», після відправлення повідомлення про зменшення лічильника скелетона віддаленого процесу, асоційованого із замісником.

В ієрархічно організованих процесах у групах необхідно використовувати більш масштабне рішення для розподіленого збирання сміття, тому що

низькорівневі групи збирають своє сміття, а аналіз посилань між групами залишають на групи більш високого рівня. Оскільки кількість об'єктів у низькорівневих групах, які потрібно відслідковувати, зменшується, групи верхнього рівня здебільшого працюють зі схожою кількістю об'єктів, розподілених більшою мережею.

5.4 Висновки

1. Розрізняють такі основні типи імен, які використовують для посилань на сутності:

а) адреса сутності – ім'я точки доступу, асоційоване із сутністю;

б) ідентифікатор, який має такі властивості:

– кожна сутність має тільки один ідентифікатор,

– ідентифікатор указує на єдину сутність і не може бути перепризначений іншій;

в) імена, зручні для сприйняття, призначені для використання людьми і є рядком символів.

2. Імена об'єднані у простори імен, які можуть бути зображені у вигляді графу іменування, у вузлах якого розташовані іменовані сутності, а мітки на ребрах є іменами, за якими ці сутності відомі. Вузли, з яких виходить кілька ребер, є наборами сутностей і відомі також як вузли напрямку. Великомасштабні графи іменування часто подаються у вигляді кореневих ациклічних напрямлених графів.

3. Графи іменування використовують для структурування імен, зручних для сприйняття. Доступ до сутності може здійснюватися за допомогою шляху. Дозвіл на доступ (дозвіл імен) імен – це процес проходження графом іменування, шукаючи компоненти, які входять у шлях, по одному за раз. Великомасштабний граф іменування реалізується за допомогою розподілу вузлів по декільком серверам імен. Визначення шляху виконується за рахунок обходу графу іменування, якщо під час пошуку розшукуваний вузол перебуває на наступному сервері імен, то пошук імен продовжується на ньому.

4. Системи іменування для імен, зручних для сприйняття, неможливо використовувати для високомобільних сутностей. Локалізація мобільних сутностей більш успішно може бути здійснена за допомогою незалежних від місця розташування ідентифікаторів.

5. Розрізняють чотири основні підходи до локалізації мобільних сутностей.

Перший підхід полягає у використанні широкомовних або групових розсилань. Ідентифікатор сутності відправляється широкомовним посиланням кожному процесу в розподіленій системі, процес пропонує точку доступу у відповідь на надання йому адреси цієї точки доступу. Очевидно, що цей підхід має обмежену масштабованість.

Другий підхід полягає в пересиланні вказівників. Щоразу сутність, переміщуючись в інше місце, залишає замість себе вказівника, який інформує про те, куди вона перемістилася. Локалізація сутності вимагає обходу ланцюжка вказівників, які пересилаються. Щоб усунути довгий ланцюжок вказівників, важливо зменшити довжину ланцюжка після переміщення.

Третій підхід полягає у створенні бази сутності. Щоразу сутність, переміщуючись в інше місце, повідомляє про це свою базу. В разі локалізації сутності передусім про поточну локалізацію запитується її база.

Четвертий підхід полягає у побудові ієрархічного дерева пошуку. Мережа розбивається на зони, які не перекриваються (домени) та можуть групуватися в домени верхнього рівня і так далі. Наявний один домен верхнього рівня, який охоплює всю мережу. Кожний домен на будь-якому рівні має асоційований з ним вузол напрямку. Якщо сутність перебуває в домені *D*, то вузол напрямку в домені наступного рівня містить вказівник на *D*; вузли напрямку найнижчого рівня – адреси сутностей; вузол напрямку найвищого рівня – відомості про всі сутності.

6. Сутності, локалізація яких більше не потрібна, можуть бути видалені. Важлива мета використання імен у розподілених системах – створення посилань на сутності так, щоб сутності, на які немає посилань, видалялися автоматично. Таке збирання сміття вимагає підрахунку посилань або трасування.

7. У разі підрахунку посилань сутність обчислює кількість створених на неї посилань. Коли лічильник досягне нуля, сутність можна видаляти. Крім підрахунку посилань, можна також підтримувати список посилань на процеси, які посилаються на сутність. Список посилань більш стійкий, ніж лічильник посилань, але погано масштабується.

8. У методах трасування всі сутності прямо або опосередковано посилаються на заданий набір кореневих сутностей, позначених як доступні, недоступні сутності видаляються. Розподілене трасування важке, оскільки потрібно перевірити всі сутності в системі. Рішення цієї проблеми різноманітні, але зазвичай у їх основу покладено традиційні збирачі сміття, характерні для однопроцесорних систем.

21.2. **Запитання для самоконтролю**

1. Визначте поняття «ім’я», «сутність», «адреса», «точка доступу».
2. Визначте поняття «ідентифікатор», «правильний ідентифікатор».
3. Визначте поняття «глобальне ім’я», «локальне ім’я». Навіщо їх використовують і яка між ними відмінність?
4. Опишіть процес ідентифікації імен. Де цей процес використовують у розподілених системах?
5. Опишіть процес рекурсивної ідентифікації імен. Як він виконується?
6. Опишіть процес ітеративної ідентифікації імен. Як він виконується?
7. У чому полягає основний недолік рекурсивної ідентифікації імен?
8. Які переваги рекурсивної ідентифікації імен?
9. Де використовують і як працює протокол LDAP?
10. Як визначається ефективність широкомовного розсилання в розподіленій системі?
11. Опишіть процедуру групового розсилання в розподіленій системі.
12. Локальне кешування адрес.
13. Опишіть процедуру простого підрахунку посилань.
14. Опишіть процедуру зваженого підрахунку посилань.

15. Де і чому використовують непрямі посилання й як виконується підрахунок поколінь посилань?

16. Що таке «ідемпотентні операції» та як вони виконуються?

17. Де використовують і як виконується примітивне трасування в розподіленій системі?

18. Навіщо використовують і як виконується трасування в групах у розподіленій системі?

19. Опишіть базовий алгоритм збирання сміття в розподіленій системі.

20. У чому полягають основні підходи до локалізації мобільних сутностей?

6. СИНХРОНІЗАЦІЯ

6.1 Синхронізація часу

В однопроцесорних системах завдання взаємного виключення, критичних областей та інших проблем синхронізації здійснювалися з використанням загальних методів, таких як семафори та монітори. Проте ці методи не враховують специфіки розподілених систем, оскільки вони застосовують розподілення оперативної пам'яті. Наприклад, два процеси, які взаємодіють використовуючи семафор, потребують доступу до нього. Якщо обидва процеси виконуються на одній машині, то вони можуть мати спільний доступ до семафора, який зберігається, наприклад, у ядрі, роблячи системні виклики. Однак, якщо процеси виконуються на різних машинах, то цей метод не застосовують, оскільки для розподілених систем потрібні інші підходи.

Приклад. Розглянемо, як процеси синхронізуються між собою. Кілька процесів не здатні одночасно одержати доступ до спільно використовуваних ресурсів, таких як принтери, хоча мають взаємодіяти, дозволяючи один одному тимчасово одержувати ексклюзивний доступ до цього ресурсу. Водночас кілька процесів можуть час від часу потребувати в узгодження порядку проходження повідомлень, наприклад узгодження про те, що повідомлення *m1* від процесу *P* буде відправлено раніше, ніж повідомлення *m2* від процесу *Q*.

Синхронізація в розподілених системах значно складніша, ніж в однопроцесорних або мультипроцесорних системах.

Розглянемо два важливі окремі аспекти, які стосуються синхронізації, – це взаємні виключення в розподілених системах і розподілені транзакції. Розподілені взаємні виключення дозволяють спільно використовувати ресурси, запобігши можливості одночасного доступу декількох процесів. Розподілені транзакції гарантують цілісність і достовірність інформації за допомогою вдосконаленого механізму контролю за їх паралельним виконанням.

У централізованих системах час визначається однозначно. Якщо процесу необхідний час, то він здійснює системний виклик, і ядро надає йому відповідь. У розподіленій системі якщо процес *A*, а трохи пізніше і процес *B*, запи-

тує час, то значення, яке одержить процес *B,* буде більшим (або дорівнюватиме) від значення, отриманого процесом *A*, але воно не може бути і меншим.

Приклад. Для того щоб зрозуміти, до чого можуть призвести проблеми визначення глобального часу, розглянемо приклад із програмою *make* операційної системи UNIX, у якій зазвичай великі програми розбиваються на кілька файлів вихідного тексту, і внесення змін в один із цих файлів вимагає повторної компіляції не всіх, а тільки одного з них. Такий підхід значно підвищує продуктивність роботи програмістів, зокрема якщо програма містить 100 файлів, а виправлений був тільки один, то достатньо перекомпілювати лише цей файл.

Звичайна методика застосування програми *make* проста. Коли програміст закінчує свою роботу, змінивши всі файли з текстом програми, він запускає програму *make*, яка перевіряє час останньої модифікації всіх файлів з вихідним текстом і всіма об'єктними файлами програми. Якщо файл із вихідним текстом *input.c* має час останньої модифікації 21:51, а відповідний йому об'єктний файл *input.o* – 21:50, то *make* вважає, що *input.c* із часу створення *input.o* був змінений, і перекомпільовує його. Натомість, якщо *output.c* має час останньої модифікації 21:44, а *output.npo* – 21:45, то компіляція не потрібна. Таким чином, програма *make* перевіряє всі файли з вихідним текстом у пошуках тих, які потребують повторної компіляції.

Уявімо, що відбудеться, якщо в розподіленій системі немає глобального узгодження часу. Файл *output.o*, як і раніше, має позначку часу зміни 21:44, а *output.c* після створення був модифікований, але одержав позначку часу 21:43, тому що годинники на машині, де він перебуває, трохи запізнюються, як це показано на рис. 6.1. Тоді програма *make* не стане викликати компілятор, у результаті чого виконуваний файл програми міститиме суміш об'єктних файлів зі старих і нових вихідних файлів, що у процесі виконання може зумовити помилки.

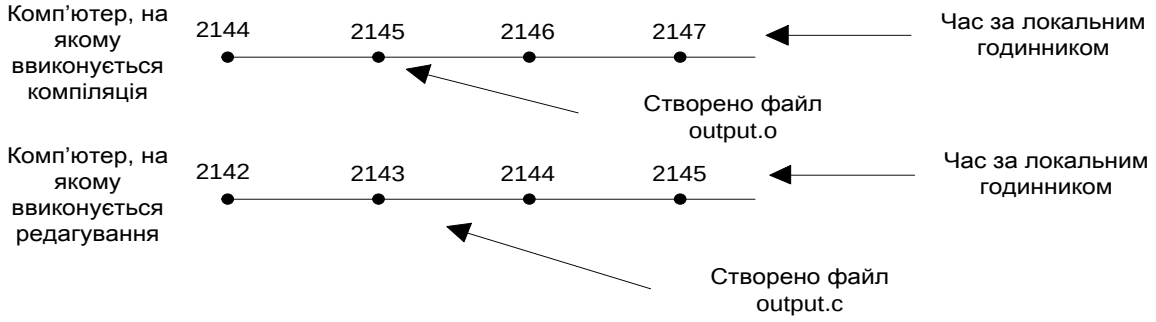


Рис. 6.1. Коли кожна з машин має власного годинника, подію, яка відбулася пізніше від іншої, може бути віднесено до більш раннього часу

6.1.1 Фізичні годинники

Майже всі комп'ютери мають **вбудовані періодичні процеси** для підрахунку часу. Незважаючи на постійне використання для цих пристроїв назви *«годинники»*, вони не є годинниками у звичайному значенні цього слова. Мабуть, більш правильним терміном був би **таймер** *(timer)* **комп'ютера**, тобто особливим чином оброблений кристал кварцу, який, перебуваючи під напругою, коливається зі сталою частотою, що залежить від властивостей кристалу, таких як спосіб розрізування і значення напруги. З кожним кристалом асоційовано два регістри: **лічильник** *(counter)* і **регістр зберігання** *(holding register).* Кожне коливання кристала зменшує лічильник на одиницю. Коли значення лічильника досягає нуля, генерується переривання й лічильник перезавантажується з регістра зберігання. Таким чином, можна запрограмувати таймер, щоб він генерував переривання 60 разів за секунду або з будь-якою іншою потрібною частотою. Кожне переривання викликається одним **тіком** таймера *(clock tick).*

Коли система завантажується вперше, вона просить користувача ввести дату й час, які перераховуються в кількість тіків, починаючи з певної стартової дати, і зберігаються в пам'яті. Багато комп'ютерів мають спеціальну мікросхему CMOS RAM із живленням від акумулятора, в результаті чого їм не потрібно вводити час у процесі кожного завантаження. Після кожного тіку таймера процедура обробки переривання додає одиницю до часу, що зберігається в пам'яті комп'ютера. Таким чином, годинники (програмні) зберігають правильний час.

Для єдиного комп'ютера та єдиних годин маленька неточність цих годин не спричинить проблеми, оскільки всі процеси в машині використовують однакові годинники.

Приклад. Якщо файл *input.c* має позначку часу 21:51, а файл *input.o* – 21:50, то *make* перекомпілює цей файл, навіть якщо годинники відстають на два розподіли і дійсний час становить 21:53 і 21:52 відповідно. Процедура *make* виконує обробку відносного часу.

Якщо розглядати декілька комп'ютерів, кожний з яких має власний годинник, то картина зміниться. Незважаючи на те, що частота кожного з кристалів зазвичай достатньо стабільна, неможливо гарантувати, що кристали на

різних комп’ютерах матимуть абсолютно однакову частоту. На практиці, коли система охоплює *n* комп'ютерів, усі *n* кристалів працюють із різною частотою, що зумовлює поступову втрату синхронізації й видачу годинниками у процесі звертання до них різних значень. Цю різницю в даних годинників називають *розсинхронізацією часу (clock skew)*, у результаті якої програми, які очікують, що час, який асоціюється з файлом, об’єктом або повідомленням, є коректним і не залежить від машини, на якій цей час визначався (тобто від використаних годинників), можуть працювати неправильно.

У деяких системах, **системах реального часу**, точність годин необхідна. Для таких систем потрібно використовувати зовнішні фізичні годинники, причому для ефективності й захищеності бажано мати кілька фізичних годинників, що зумовлює дві проблеми: по-перше, *синхронізацію їх із годинниками реального світу*; по-друге, *синхронізацію годинників один з одним.*

Розглянемо, як на практиці вимірюють час. Відколи у XVII ст. було винайдено механічний годинник, час стали визначати за результатами астрономічних спостережень. Щодня сонце встає на сході, піднімається на максимальну висоту й заходить на заході. Момент підйому сонця на максимально можливу висоту називають **сонячним переходом** (*transit of sun*), який відбувається близько полудня. Інтервал між двома послідовними сонячними переходами називають **сонячним днем** (*solar day*), який поділяють на 24 год, кожна з яких містить 3600 с. **Сонячну секунду** *(solar second)* визначають як 1/86400 сонячного дня. Геометричне представлення, необхідне для розрахунку сонячного дня, наведено на рис. 6.2.

У 40-х роках XX ст. було встановлено, що період обертання землі непостійний. Земля сповільнюється через приливне тертя й в’язкість атмосфери. На основі вивчення кілець росту древніх коралів геологи вважають, що 300 млн. років тому один рік складався з близько 400 днів, а тривалість року (час одного обертання навколо сонця) при цьому не змінилася, доба просто стала довшою. Крім цієї повільної зміни, є короткі варіації тривалості доби, які пов’язані, ймовірно, із завихреннями глибоко в земному ядрі, яке містить розплавлене залізо. Ці відкриття вимагали від астрономів розрахунку середньої тривалості доби, враховуючи дуже велику кількість днів, з усереднен-

ням результатів перед діленням їх на 86400. Отримане значення було названо **середньою сонячною секундою** (*mean solar second*).

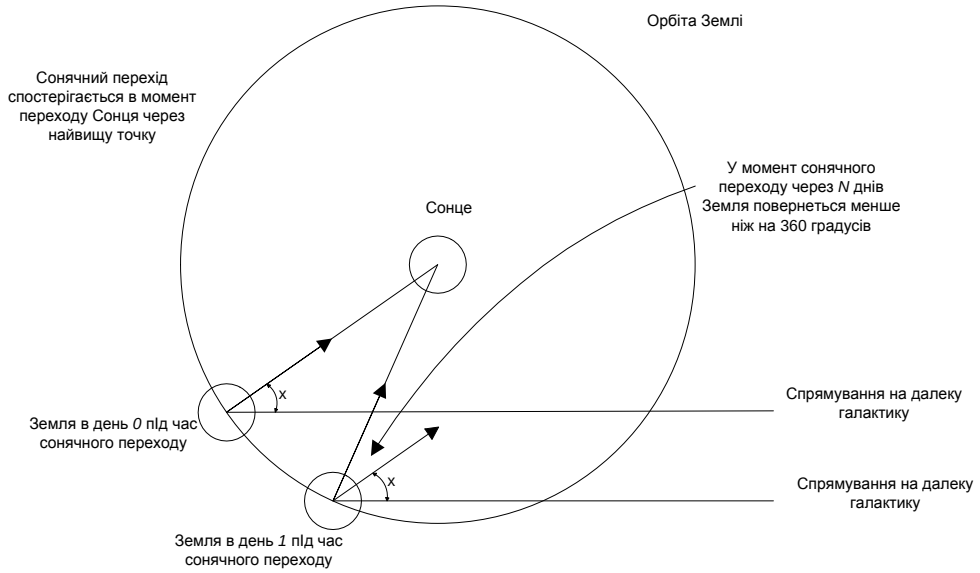


Рис. 6.2. Розрахунок середнього сонячного дня

Після винаходу 1948 року атомних годин з'явилася можливість ще більш точного вимірювання часу, незалежно від обертання й коливань Землі за допомогою підрахунку переходів атома цезію-133. Фізики замість астрономів почали зберігати час і визначили секунду як час, за який атом цезію-133 здійснить рівно 9 192 631 770 переходів. Вибір числа 9 192 631 770 прирівняв атомну секунду до середньої сонячної секунди в рік її розрахунку. Нині близько 50 лабораторій у всьому світі мають годинники на цезії-133 і періодично кожна лабораторія повідомляє Міжнародне бюро міри і ваги у Парижі, котра година на їхніх годинниках. Міжнародне бюро усереднює ці результати й визначає глобальний час за атомними годинниками *(International Atomic Time, TAI),* тобто середній час тіків годин на цезії-133, який минув від середини ночі 1 січня 1958 року (початку відліку часу) і поділений на 9 192 631 770.

Хоча час *TAI* є стабільним і доступним кожному, виникає суттєва проблема: 86 400 сонячних секунд за *TAI* нині приблизно на 3 мс менше від середнього сонячного дня, тому що середній сонячний день весь час подовжується. Використовуючи *TAI* для зберігання часу, треба розуміти, що поступово полудень буде наставати все раніше й раніше. Це стане помітно

населенню, і виникне ситуація, яка була 1582 року, коли Папа Григорій XIII оголосив, що 10 днів календаря пропускаються.

Міжнародне бюро розв'язало цю проблему, використовуючи **загублені секунди** (*leap seconds*) щоразу, коли різниця між часом *TAI* і сонячним часом зростала до 800 мс, як показано на рис. 6.3. Це дозволило перейти до системи, в основу якої покладено сталі секунди *TAI,* та дотримується відповідність із періодичністю руху сонця. Цю систему названо **універсальним погодженим часом** (*Universal Coordinated Time*) та покладено в основу всієї системи зберігання часу в наші дні, замість старого стандарту – **середнього часу за Гринвічем** (*Greenwich mean time*), який ґрунтувався на астрономічних спостереженнях і розрахунках.

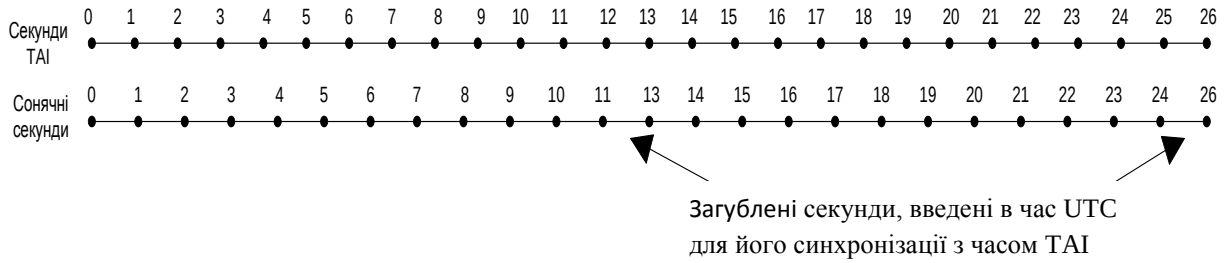


Рис. 6.3. Секунди *TAI*, на відміну від сонячних секунд, мають сталу тривалість, тому для синхронізації *UTC* і часу *TAI* використовують загублені секунди

Більшість електричних компаній поклали в основу вимірювання часу для своїх 60- або 50-герцових годин систему *UTC*. Коли міжнародне бюро повідомляє про загублену секунду, компанії, які виробляють електроенергію, змінюють частоту на 61 або 51 Гц для 60 або 50 с, щоб трохи синхронізувати годинники в тій зоні, де вони працюють. Оскільки для комп'ютера 1 с – це помітний інтервал, операційна система, якій потрібно знати точний час за кілька років, може обробляти оголошені загублені секунди за допомогою спеціального програмного забезпечення (якщо вона не використовує для визначення часу лінію живлення, що зазвичай дуже неточно). Загальна кількість загублених секунд, введених під час *UTC*, до сьогодні становить близько 30.

Для надання часу *UTC* тим, кому необхідний точний час, Національний інститут стандартного часу (*National Institute of Standards and Technology, NIST*) має короткохвильову радіостанцію з позивними *WWV* із форту Коллінз

(Fort Collins), штат Колорадо. Радіостанція *WWV* широкомовно розсилає короткий імпульс на початку кожної секунди *UTC*. Точність самої радіостанції *WWV* дорівнює близько ±1 мс, але через різні атмосферні флуктуації довжина сигналу може змінюватися, тому на практиці точність є не більшою ніж ±10 мс. В Англії станція *MSF*, яка працює з Регбі *(Rugby)*, район Варвикшир *(Warwickshire)*, надає аналогічну службу. Існують також станції й у деяких інших країнах.

Деякі супутники Землі також надають служби *UTC*. Робочий супутник геостаціонарного оточення *(Geostationary Environment Operational Satellite, GEOS)* може надавати час *UTC* із точністю до 0,5 мс, а інші – і з більш високою точністю.

Використання як короткохвильових радіосигналів, так і супутникових служб, вимагає точних відомостей про відносне положення відправника й одержувача, щоб компенсувати затримку поширення сигналу.

5.1.1. Алгоритми синхронізації часу

Якщо одна машина має приймач *WWV*, то її завданням є підтримка синхронізації з нею всіх інших машин. Якщо приймачів *WWV* немає на жодній з машин, то кожна з них розраховує власний час, і тоді завданням буде по можливості синхронізувати їх між собою. Для здійснення синхронізації було запропоновано багато алгоритмів.

Всі алгоритми мають базову модель системи. Вважається, що кожна машина має таймер, який ініціює переривання H разів за секунду. Коли цей таймер спрацьовує, обробник переривань додає одиницю до програмних годинників, які зберігають кількість тіків (переривань), починаючи з певного моменту в минулому, про який була попередня домовленість. Будемо вважати, що можна отримати C – значення цих годин, тобто коли час UTC дорівнює t, значення годин машини p дорівнюватиме $C\,p\,(t)$. В ідеальному випадку можемо вважати, що для всіх p і всіх t, тобто $\frac{d}{dt}$ є точною одиницею.

Реальні таймери не генерують переривання точно H разів за секунду. Теоретично таймер з $H = 60$ має генерувати 216 000 тіків за годину. На прак-

тиці відносна похибка, яка допускається в сучасних мікросхемах таймерів, становить близько 10^{-5}, тобто конкретна машина може подати значення в діапазоні від 215998 до 216002 тіків за годину.

Нехай константа p становить $-p<dC/dt< 1+p$. У цих межах таймер можна вважати працездатним. Константа p залежить від виробника і відома під назвою «**максимальна швидкість дрейфу** *(maximum drift rate) »*. Співвідношення між часом годинників, які відстають, правильних і годинників, які поспішають, ілюструє рис. 6.4.

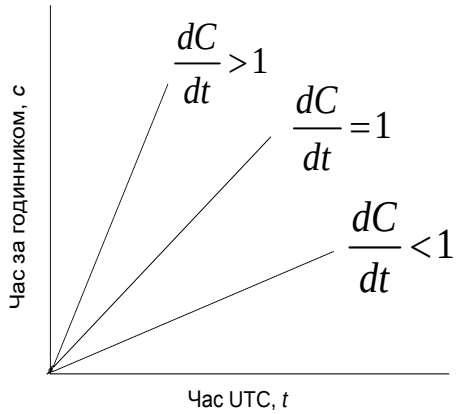


Рис. 6.4. Співвідношення між часом за годинником і часом UTC

Якщо два годинники UTC йдуть у різні боки за час Δt після синхронізації, то різниця між їх показаннями може бути не більш ніж $2p - \Delta t$. Якщо розробники операційної системи хочуть гарантувати, що жодна пара годин не зможе розійтися більш ніж на δ, синхронізацію годин слід виконувати не рідше, ніж кожні $\delta/2p$ с. Алгоритми синхронізації розрізняються точністю визначення моменту здійснення повторної синхронізації.

Алгоритм Крістиана. Розпочнемо з алгоритму, який грунтується на роботах Крістиана та використовують для систем, у яких одна з машин має приймач *WWV*, і потрібно синхронізувати всі інші машини за нею. Назвемо машину з приймачем *WWV* сервером часу (*time server*). Періодично, гарантовано не рідше ніж кожні $\delta/2p$ с, кожна машина надсилає серверу часу повідомлення, у якому запитує поточний час, потім так швидко, як це можливо, відповідає повідомленням, які містять значення її поточного часу, C_{UTC}.

У разі першого наближення, коли відправник отримує відповідь він може просто налаштувати у своєму годиннику значення C_{UTC} , але такий алгоритм має

дві проблеми. Головна проблема полягає в тому, що час ніколи не відраховують назад. Якщо годинник відправника поспішає, то значення C_{UTC} може виявитися меншим за поточне значення C_{UTC} відправника. Просте використання C_{UTC} може зумовити суттєві помилки, пов'язані з тим, що деякі файли скомпільовані, наприклад після того, як було змінено час, а фактично позначені більш раннім часом, ніж модифіковані вхідні тексти, які виправлялися до зміни часу. З огляду на це зміни потрібно вносити поступово, зокрема у такий спосіб, який наведено у прикладі.

Приклад. Таймер налаштовано таким чином, що він генерує 100 переривань за секунду. В нормальному стані кожне переривання додаватиме до часу по 10 мс; у разі запізнення годинника – лише по 9 мс. З огляду на це годинник потрібно виправити так, щоб додавати під час кожного переривання по 11 мс, а не перемикатися одночасно.

Менш суттєва проблема полягає в тому, що перенесення відповідного повідомлення із сервера часу відправника потребує ненульового часу. Ця затримка може бути дуже великою та залежати від завантаження мережі. За алгоритмом Крістиана, способом розв'язання цієї проблеми є вимірювання цієї величини. Відправнику достатньо акуратно записати інтервал часу між відправленням запита й надходженням відповіді. Визначений час (початковий *To* та кінцевий T_1 вимірюються за одним годинником, а отже, інтервал буде відносно точно виміряний навіть за умови, що годинник відправника має деякі відмінності з *UTC*.

За умови браку додаткової інформації найкращим наближенням буде час проходження повідомлення $(T_1 - T_0) / 2$. Після отримання відповіді, щоб установити приблизний поточний час сервера, значення часу, яке міститься в повідомленні, потрібно збільшити на це число. Якщо теоретично мінімальний час проходження відомий, то потрібно розрахувати інші параметри часу.

Оцінка часу може бути покращена, якщо відомо, скільки приблизно часу сервер обробляє переривання та працює з повідомленням, яке надійшло. Позначимо час обробки переривання I, тоді значення інтервалу часу між T_0 та T_1, який затрачено на проходження повідомлення, дорівнюватиме $T_1 - T_0 - I$. Найкращою оцінкою часу проходження повідомлення в один бік буде половина цього числа. Є системи, в яких повідомлення з A до B постійно

пересилається одним маршрутом, а з B до A – другим. За таких умов отримаємо різний час проходження двох повідомлень за різними маршрутами.

Для підвищення точності в алгоритмі Крістиана запропоновано виконувати не одне вимірювання, а серію. Всі виміри, в яких різниця $T_1 - T_0$ перевищує деяке граничне значення, відкидаються як результат перевантаження мережі, тобто як недостовірні. Оцінювання виконують за вимірами, які залишилися та які можна усереднити для отримання найкращого значення. Натомість повідомлення, які надійшли швидше за всіх, можна розглядати як найточніші, оскільки вони ймовірно потрапили в час найменшого навантаження, тому найбільш точно відображають чистий час проходження.

Алгоритм Берклі. В алгоритмі Крістиана сервер часу є пасивним, а інші машини періодично запитують у нього час, тому вся його робота – це відповіді на запити. В операційній системі *UNIX*, розробленій університетом Берклі (*Berkeley*) обрано протилежний підхід. У цій операційній системі сервер часу, а фактично демон часу, є активним, він періодично опитує кожну машину, який час показує її годинник. На основі відповідей він обчислює середній час і пропонує всім машинам установити на їх годинниках новий час або сповільнити час, доки не буде досягнуто необхідного зменшення часу на годинниках, які поспішають уперед. Цей метод можна використовувати для систем, які не мають машин з приймачем *WWV*. Час демона може періодично виставляти вручну оператор.

Усереднюючі алгоритми. Описані алгоритми централізовані, звідки випливає низка недоліків. Відомі також і децентралізовані алгоритми. Один з класів алгоритмів децентралізованої синхронізації годинників працює на основі поділу часу на синхронізаційні інтервали фіксованої тривалості, коли z-й інтервал починається в момент часу $T_0 + iR$ і триває до $T_0 + (i + 1)R$, де T_0 – попередньо узгоджений минулий момент, а R – параметр системи. На початку кожного інтервалу кожна машина виконує широкомовну розсилку значення поточного часу на своєму годиннику. Оскільки годинники на різних машинах ідуть з трохи різною швидкістю, ці широкомовні розсилки будуть виконані не одночасно. Після розсилки машиною свого часу вона

запускає локальний таймер та починає збирати всі інші широкомовні пакети протягом деякого інтервалу *S*. Коли буде зібрано всі широкомовні пакети, запускається алгоритм обчислення за ними нового часу. Найпростіший алгоритм полягає в усередненні значень годинників усіх інших машин. Незначні його варіації полягають передусім у відкиданні *m* найбільших і *m* найменших значень та усередненні тих, які залишилися. Відкидання крайніх значень можна розглядати як самозахист від *m* неправильних годинників. Інша реалізація алгоритму полягає в тому, щоб спробувати відкоригувати кожне з повідомлень, додаючи до нього оцінку часу проходження від джерела на основі знання топології мережі або вимірювання часу проходження місяця.

Розглядають низку додаткових алгоритмів. Один з найбільш часто використовуваних алгоритмів у Internet – це протокол мережного часу (*Network Time Protocol, NT, NTP*) відомий тим, що дозволяє забезпечити точність (глобальну) 1-50 мс, яка досягається за рахунок використання вдосконалених алгоритмів синхронізації годинників.

Множинні зовнішні джерела точного часу. Для систем, яким необхідна особливо точна синхронізація за *UTC*, можна запропонувати використання декількох приймачів *WWV, GEOS* або інших джерел *UTC*. Однак, через неточність самих джерел часу та флуктуації на шляху сигналу операційні системи мають установити інтервал, у який потрапить *UTC*. Зазвичай різноманітні джерела точного часу є причиною різноманіття діапазонів, і машини, до яких їх приєднано, мають дотримуватися певної загальної угоди, щоб досягти якої кожний процесор із джерелом *UTC* може періодично здійснювати широкомовну розсилку своїх даних, наприклад, точно на початку кожної хвилини за *UTC*. Жодний процесор не отримує пакетів значень часу миттєво. Затримка між відправленням та надходженням залежатиме від довжини кабелю та кількості маршрутизаторів, через які має пройти пакет. Значення затримки різні для кожної пари (джерело *UTC*, процесор). Впливатимуть також фактори, такі як затримка через колізії, коли декілька машин намагатимуться надсилати щось по *Ethernet* в один момент. Більш того, якщо процесор обробляє попередній пакет, то він, імовірно, не зможе

переглянути пакет значень часу протягом установленої кількості мілісекунд, що додатково зменшить вірогідність отримання точного часу.

5.1.2. Використання синхронізованих годин

За минулі кілька років стало легкодоступним необхідне обладнання й програмне забезпечення, призначене для синхронізації годин у глобальному масштабі (тобто в усьому *Internet*), завдяки чому можна синхронізувати мільйони годинників з точністю до декількох мілісекунд за *UTC*. Почали з'являтися нові алгоритми, які використовують ці синхронізовані годинники. Один з прикладів стосується того, як домогтися доставки серверу не більше одного повідомлення, навіть у разі збоїв. Традиційний підхід полягає в тому, що кожному повідомленню надається унікальний номер повідомлення, а кожен сервер зберігає всі номери повідомлень, які він отримав, щоб можна було відрізнити нове повідомлення від повторного. Проблема цього алгоритму полягає в тому, що у разі збоїв та перезавантаження сервера він губить таблицю номерів повідомлень. Неясно також, скільки часу зберігати номери повідомлень.

З часом традиційний алгоритм модифікується в такий спосіб. Кожне повідомлення має ідентифікатор зв'язку (обраний відправником) і мітку часу. Для кожного з'єднання сервер вносить у таблицю останню отриману мітку часу. Якщо яке-небудь вхідне повідомлення має мітку часу меншу за збережену в таблиці для цього з'єднання, то повідомлення вважається дублікатом і не розглядається.

Щоб можна було видалити стару мітку часу, кожен сервер постійно підтримує глобальну змінну *G*, яка визначається в такий спосіб:

G = CurrentTime – MaxLifetime – MaxClockSkew,

де *MaxLifetime* – максимальний час життя повідомлення, а *MaxClockSkew* вказує на те, наскільки може відрізнятися час годинників від часу *UTC*.

Будь-яка мітка часу, старша за *G,* може бути вилучена з таблиці, оскільки всі повідомлення, старші за *G,* уже пройшли. Нехай вхідне повідомлення має невідомий ідентифікатор зв'язку, його буде прийнято, якщо його мітка часу більш рання, ніж *G,* і відкинуто, якщо вона більш пізня, оскільки будь-яке

більш пізнє повідомлення – це, безсумнівно, **дублікат**. Фактично, *G* – це сума номерів усіх старих повідомлень.

У разі збою і подальшого перезавантаження сервера він завантажує значення *G* з файлу, збереженого на диску, і збільшує його під час періоду відновлення, у результаті чого кожне повідомлення, яке могло бути прийняте до збою, відкидається. Деякі нові повідомлення можуть бути відкинуті помилково, але за дотримання всіх умов алгоритм підтримує принцип «не більше одного повідомлення».

5.2. Логічні годинники

Під час колективної роботи користувачів необхідно, щоб усі машини використовували однаковий час, причому не дуже важливо, щоб цей час збігався з реальним часом, який щогодини повідомляють по радіо, тому для деякого класу алгоритмів така внутрішня несуперечність має набагато більше значення, ніж те, наскільки їх час є близьким до реального. У таких алгоритмах використовуються **логічні годинники** *(logical clocks).* Так для роботи програми *make* достатньо, щоб усі машини вважали, що зараз 10:00, навіть якщо насправді 10:02.

Лампорт *(Lamport)* показав, що хоча синхронізація годин можлива, вона не обов'язково має бути абсолютною. Якщо два процеси не взаємодіють, немає потреби в тому, щоб їх годинники були синхронізовані, оскільки відсутність синхронізації залишиться непоміченою в системі й не створить проблем. Крім того, має значення не точний час виконання процесів, а **його порядок.**

5.2.1. Мітка часу Лампорта (відмітки часу)

Для синхронізації логічних годин Лампорт запропонував відношення під назвою *«відбувається раніше».* Вираз $a \rightarrow b$ читається як «*a* відбувається раніше за *b*» й означає, що всі процеси виконуються за правилом: першою відбувається подія *a*, а пізніше – подія. Відношення «відбувається раніше» безпосередньо здійснюється за таких умов: якщо *a* і *b* – події, які відбуваються в одному процесі, *a* відбувається раніше, ніж *b*, то відношення $a \rightarrow b$ істинно.

Якщо *a* – це подія відсилання повідомлення одним процесом, а *b* – подія одержання того ж повідомлення іншим процесом, то відношення також істинно.

Повідомлення *не може бути отримане до відсилання* або навіть *у той самий момент*, коли його було відправлено, оскільки для пересилання необхідний кінцевий ненульовий час.

Відношення «відбувається раніше» є транзитивним, тобто якщо $a \rightarrow b$ і $b \rightarrow c$, то виконується умова $a \rightarrow c$. Якщо дві події *x* і *y*, відбуваються в різних процесах, які не обмінюються повідомленнями (ні безпосередньо, ні через третій процес), то відношення $x \rightarrow y$ не є істинним, втім як і $y \rightarrow x$. Такі події називають *паралельними (concurrent)*. У такому разі це означає, що не можливо або не потрібно знати, де і яка із цих подій відбулася.

Для впорядкування процесів необхідний спосіб вимірювання часу кожної події, що дозволив би поставити у відповідність кожній події *a* **мітку часу** *C(a)*, яка підійшла б усім процесам та за умови $a \rightarrow b$ виконувалося співвідношення $C(a)<C(b)$, тобто якщо *a* і *b* – дві події одного процесу й *a* відбувається раніше ніж *b*, то $C(a)<C(b)$. Наприклад, якщо *a* – це відправлення повідомлення одним процесом, а *b* – отримування цього повідомлення іншим процесом, то значення *C(a)* і *C(b)* мають задовольняти відношенню *C(a)*. Крім того, час за годинниками *C* завжди йде вперед (збільшується). Корегувати час слід лише за допомогою додавання до нього додатного значення, а не його віднімання.

Розглянемо алгоритм Лампорта, який застосовують для присвоєння часу подіям. Три процеси, подані на рис. 6.5 а, запущено на різних машинах, кожна з яких має власні годинники і швидкість роботи. Коли годинники процесу *0* встановлять мітку часу *6*, для процесу *1* вони покажуть *8*, а для процесу *2 – 10*. Усі годинники йдуть зі сталою швидкістю, але значення їх швидкості різні через відмінність у кристалах.

У момент *6* процес *0* надсилає повідомлення *A* процесу *1*. Коли воно надійде у процес *1*, його годинник показуватиме *16*. Якщо повідомлення міститиме час відправлення *6*, процес *1* дійде висновку, що на пересилання пішло *10* тіків, що є цілком коректним значенням. Відповідно до цього повідомлення

A від процесу *1* буде доставлено процесу 2 за *16* тіків, що теж є коректним значенням.

Повідомлення *C* від процесу *2* до процесу *1* буде надіслано у момент часу, зафіксований на мітці *60*, і досягне мети на мітці *56*. Повідомлення *D* від процесу *1* до процесу до *0* буде відправлено на мітці *64* і буде доставлено на відмітці *54*. Таке співвідношення часових міток є некоректним.

Рішення, знайдене Лампортом, побудовано за принципом «відбувається раніше». Оскільки повідомлення *C* надсилається на відмітці *60*, воно має бути доставленим на мітці 61 або більшій, для цього кожне повідомлення містить час відправлення за годинниками відправника. Коли повідомлення доставляється, але годинники одержувача показують час, більш ранній за час відправлення, одержувач швидко підводить свої годинники так, щоб вони показували час на одиницю більше від часу відправлення, тоді (рис. 6.5, *б*) повідомлення надходить на відмітці часу *61*, а повідомлення *D* – на відмітці *70*.

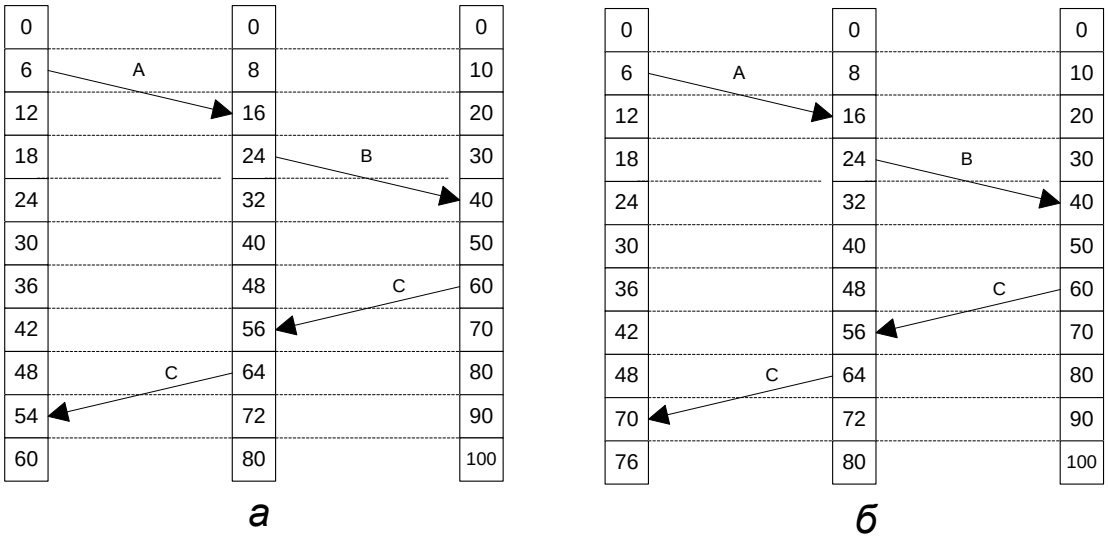


Рис. 6.5. Три процеси з власними годинниками: (а) – які ходять із різною швидкістю; (б) – підстроювання годин за алгоритмом Лампорта

Цей алгоритм задовольняє вимогам за глобальним часом з одним незначним доповненням, яке полягає в тому, що між будь-якими двома подіями годинник має «проходити» як мінімум один раз. Якщо процес надсилає або приймає в короткому сеансі два повідомлення, то він має функціонувати так, щоб його годинник встиг «пройти» хоч один раз у проміжку часу між надходженням повідомлень.

У деяких випадках бажано задовольняти додатковій вимозі: ніякі два повідомлення не повинні прийматися одночасно. Щоб виконати цю вимогу, можемо додати номер процесу, в якому відбуваються події, праворуч від мітки часу й відокремити його від цілої частини десятковою точкою. Таким чином, якщо у процесах *1* й *2* у момент часу *40* відбуваються події, то перший з них процес буде відбуватися в *40.1*, а другий у *40.2*.

Використовуючи цей спосіб, одержуємо можливість визначати час для всіх подій у розподіленій системі, які підпадають під такі умови:

- Якщо *a* відбувається раніше *b* в одному процесі, то $C(a) < C(b)$.
- Якщо *a* і *b* є відправленням й одержанням повідомлення відповідно, то маємо $C(a) < C(b)$.

Цей алгоритм дає можливість повністю впорядкувати всі події в системі, у той час як інші розподілені алгоритми для усунення неоднозначностей вимагають упорядкованості, що є достатньо складним заданням.

5.2.2. Векторна мітка часу

Мітка часу Лампорта впорядковує всі події в розподіленій системі в єдину послідовність: якщо подія *a* відбувається раніше події *b*, то *a* перебуває перед *b*, тобто $C(a) < C(b)$. Оцінювання за допомогою міток часу Лампорта не використовує взаємозв'язку між двома повідомленнями *a* і *b*, просто порівнює їх тимчасові мітки, відповідно $C(a)$ і $C(b)$. Інакше кажучи, якщо $C(a) < C(b)$, це не означає, що подія *a* дійсно відбулася раніше, ніж подія *b*.

Щоб зрозуміти цю особливість, розглянемо систему розсилання повідомлень, одним з найбільш поширених прикладів якої є служба електронних дошок повідомлень в Internet, мережні новини. Користувачі, а отже процеси, обирають певні дискусійні групи. Листи в такій групі розсилаються за допомогою **групового розсилання** всім членам групи. Щоб гарантувати, що відповіді надсилаються після листів, на які відповідали, використаємо описану **схему повністю впорядкованого групового розсилання**, яка не означає, що якщо повідомлення *B* надійшло після повідомлення *A*, то *B* є відгуком на те, що було надіслано в повідомленні *A*. Насправді повідомлення *A* і *B* можуть

бути абсолютно незалежними одне від одного. Повністю впорядковане групове розсилання занадто суворе для цього випадку.

Мітки часу Лампорта не можуть вловити причинно-наслідковий зв’язок. Наприклад, одержання листа за причинно-наслідковим зв'язком завжди випереджає відсилання відповіді на нього. З огляду на це, якщо у групі процесів підтримуються причинно-наслідкові зв’язки, то відповідь на лист завжди має одержуватися після одержання цього листа. Якщо два листи, або відповіді на них, незалежні один від одного, то порядок їх одержання зазвичай не важливий.

Причинно-наслідкового зв’язку можна дотримуватися за допомогою **векторних міток часу** *(vector timestamps)*. Векторна відмітка часу $\boldsymbol{V_T(a)}$, надана події *a*, має таку властивість*: якщо* $V_T(a) < V_T(b)$ *для події b, то подія a є причиною попередньої події b.* Векторні відмітки часу створюються за рахунок приписування кожному процесу *P* вектора з такими властивостями:

1. $V_j(i)$ – це кількість подій, які відбулися з процесом *P* до цього моменту.
2. Якщо $V_j(i) = S$, то процес P_S знає, що з процесом *P* відбулося *S* подій.

Перша властивість підтримується за рахунок збільшення $V_j(i)$ на одиницю під час кожної нової події, що відбувається у процесі *P*, а друга – вкладанням векторів у повідомлення, які відсилаються. Коли процес P_S надсилає повідомлення *T*, він пересилає разом з ним і його поточний вектор як відмітку часу V_t. Таким чином, одержувач стає проінформованим про номер повідомлення, яке зумовило активність процесу P_S. Крім того, одержувач повідомляється про те, скільки повідомлень від інших процесів має надійти до нього до того моменту, як процес P_j надіслав йому повідомлення *T*. Інакше кажучи, мітка часу V_t повідомлення *T* визначає для одержувача, скільки подій в інших процесах мають випереджати *T* і з яким з них у повідомлення *T* може бути причинно-наслідковий зв'язок. Коли процес P_j одержує *T*, це спонукає його встановити кожен елемент $V_j(k)$ у власному векторі як максимальне значення $\{max\{V_j(k), V_t(k)\}$. Вектор тепер відображає кількість повідомлень, які має одержати процес *P* і які передували відправленню повідомлення *T*. Потім елемент $V_j(i)$ збільшується на одиницю, відображаючи факт одержання повідомлення *T*, як наступного повідомлення від процесу *P*.

Векторна мітка часу може використовуватися для доставки повідомлень тільки без порушення причинно-наслідкового зв'язку. Розглянемо приклад з електронною дошкою оголошень. Коли процес P_i відправляє лист, він здійснює множинне розсилання цього листа у вигляді повідомлення A з позначенням часу $v_t(a)$, що дорівнює v_i. Коли інший процес P_j одержує A, він виправляє власний вектор, установлюючи $v_j(i) > v_t(a)$.

P_j надсилає відповідь на цей лист за допомогою множинного розсилання повідомлення r з відміткою часу $v_t(r) = V_j$. Відзначимо, що $v_t(r) > v_t(a)$. Якщо вважати зв'язок надійним, то обидва ці повідомлення (a, що містить лист, і r, що містить відповідь) досягнуть іншого процесу P_k. Оскільки не зроблено ніяких припущень щодо черговості доставки повідомлень, повідомлення r може надійти до процесу P_k раніше, ніж повідомлення a. Одержавши r, P_k перевіряє відмітку часу $v_t(r)$ і вирішує відкласти доставку до того моменту, поки не будуть прийняті всі повідомлення, які йдуть перед T, тобто повідомлення r буде доставлено тільки у разі виконання двох умов:

1. $v_t(r) = v_k(j)+1$;
2. $v_t(r) < v_k(i)$ для всіх $i \neq j$.

Перша умова означає, що r – наступне повідомлення, одержуване P_k від процесу P_r, друга умова – що P_k бачить такі самі повідомлення, які бачив процес P_j, відправляючи повідомлення r, тобто процес P_k уже одержав повідомлення a.

5.3. Глобальний стан

Для колективної роботи в розподіленому інформаційному середовищі потрібно знати глобальний стан, відповідно до якого розподілена система функціонує. Глобальний стан **(global state)** розподіленої системи містить локальні стани кожного процесу разом з повідомленнями, які перебувають у процесі пересилання (тобто є відправленими, але ще не доставленими). Для розподіленої бази даних у нього можуть входити лише ті записи, з яких складається частина бази даних, а тимчасові записи, які використовуються для обчислень, можна вилучити.

У простому способі безпосереднього запису глобального стану розподіленої системи вводять поняття «**розподілений знімок стану**» *(distributed snapshot)*, яке відображає стан, у якому перебувала розподілена система, зокрема **несуперечливий глобальний стан.** Якщо у знімку записано, що процес *P* одержав повідомлення *A* від процесу *Q*, то там також має бути записано, що процес *Q* відправив це повідомлення *A*. Інакше знімок стану містив би запис про повідомлення, які були прийняті, але ніколи ніким не відправлялися. Проте допускається зворотна ситуація, коли процес *Q* надіслав повідомлення, а процес *P* його ще не одержав.

Поняття «глобальний стан» можна зобразити графічно за допомогою **зрізу** *(cut)*, де зокрема на рис. 6.6, а показано несуперечливий зріз пунктирною лінією, він перетинає осі часу процесів – *P1*, *P2* і *P3*. Зріз відображає останню записану подію для кожного з процесів, у чому легко переконатися, оскільки для всіх записів про повідомлення, які надійшли, зафіксовано відповідні події їх відправлення. Суперечливий зріз показано на рис. 6.6 б), на якому надходження повідомлення *m2* у процес P_3 було записано, але у знімку стану не зафіксовано відповідної події його відправлення.

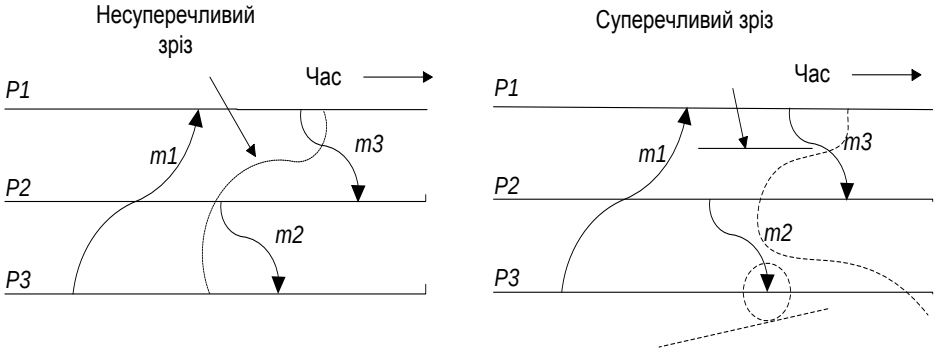


Рис. 6.6. Знімки стану: а - несуперечливий зріз; б - суперечливий зріз

Щоб спростити пояснення алгоритму створення розподіленого знімка стану припустимо, що розподілену систему можна подати у вигляді набору процесів, з'єднаних один з одним за допомогою однонапрямних прямих комунікаційних каналів. Так процеси можуть на першому етапі взаємодії встановлювати з'єднання TCP.

Ініціювати алгоритм може будь-який процес. Ініціювавши одержання розподіленого знімка стану процес, скажімо *P*, починає записувати власний

локальний стан, потім надсилає маркер за кожним зі своїх вихідних каналів, даючи зрозуміти, що одержувач цього повідомлення має взяти участь у записуванні глобального стану.

Коли процес *Q* одержує з вхідного каналу *C* маркер, його подальші дії залежать від того, записав він уже свій локальний стан чи ні. Якщо ні, то він спочатку записує свій локальний стан, а потім також розсилає маркери в усі вихідні канали. Якщо *Q* уже записав свій стан, то маркер, який надійшов з каналу *C*, показує, що *Q* має записати стан каналу, який складається з послідовності повідомлень, прийнятих процесом *Q* з моменту запису ним свого локального стану й до надходження маркера. Запис цього стану ілюструє рис. 6.7.

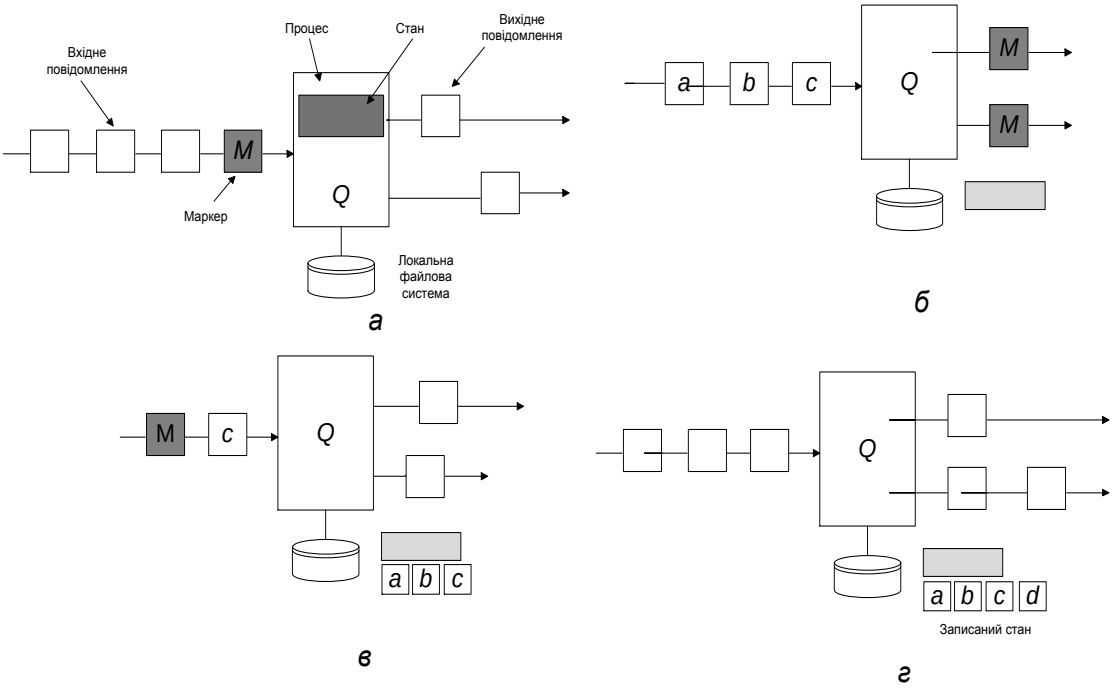


Рис. 6.7. Організація процесу й каналів у розподіленому знімку стану:
а) процес Q одержує маркер уперше й записує свій локальний стан; б) процес Q записує всі вхідні повідомлення; в) процес Q одержує маркер із вхідного каналу; г) процес Q закінчує запис стану вхідного каналу

Процес повідомляє про те, що він закінчив свою частину алгоритму, після одержання й обробки маркера від кожного зі своїх вхідних каналів. У цей момент записані локальний стан і стан для кожного із вхідних каналів разом надсилаються, наприклад, процесу, який ініціював створення знімка стану та може потім проаналізувати поточний стан. Відзначимо, що

під час роботи такого алгоритму розподілена система в цілому продовжує працювати у звичайному режимі.

Оскільки алгоритм може бути ініційований будь-яким процесом, одночасно може створюватися кілька знімків стану, тому маркер повинен мати **ідентифікатор і номер версії процесу**, який ініціював алгоритм. Лише після того, як процес одержить цей маркер за усіма вхідними каналами, він може закінчити побудову своєї частини асоційованого з маркером знімка стану.

Приклад. Визначення моменту завершення. Якщо процес *Q* одержує маркер, який запитує знімок стану вперше, то він розглядає процес, який надіслав цей маркер, як попередника. Коли *Q* завершує створення своєї частини знімка стану, він надсилає своєму попередникові повідомлення «*ГОТОВО*». Рекурсивним способом, коли ініціатор створення розподіленого знімка стану одержує повідомлення «*ГОТОВО*» від усіх своїх спадкоємців, він знає, що знімок стану повністю готовий.

Знімок стану може показати глобальний стан, перебуваючи у якому повідомлення ще перебувають у стані пересилання. Зокрема, якщо є запис процесу про те, що ним одержано повідомлення за одним зі своїх вхідних каналів у період між записом локального стану й одержанням маркера із цього каналу, то неможливо зробити висновки про закінчення розподілених обчислень, оскільки це повідомлення може зумовити нові повідомлення, які вже не будуть частиною знімка стану.

У такому разі потрібно одержати знімок стану з усіма порожніми каналами. Коли процес *Q* закінчує створення своєї частини знімка стану, він надсилає попередникові або повідомлення «*ГОТОВО*», або повідомлення «*ПРОДОВЖИТИ*». Повідомлення «*ГОТОВО*» повертається лише в тому разі, якщо виконані такі умови:

1. Всі попередники *Q* повернули повідомлення «*ГОТОВО*».
2. Між записом свого стану й одержанням маркера процес *Q* не приймав повідомлень за жодним зі своїх вхідних каналів.
3. В усіх інших випадках процес *Q* надсилає попередникові повідомлення «*ПРОДОВЖИТИ*».

Ініціатор створення знімка стану, скажімо процес *P*, може одержувати від кожного зі своїх спадкоємців або повідомлення «*ПРОДОВЖИТИ*», або повідомлення *ГОТОВО*. Якщо були отримані тільки повідомлення «*ГОТОВО*», то це означає, що звичайного обміну повідомленнями не відбувається, а отже, обчислення завершені, інакше процес *P* ініціює створення нового знімка стану і продовжує доти, поки не одержить повного комплекту повідомлень «*ГОТОВО*».

На сьогодні розроблено низку рішень для визначення моменту завершення обчислень.

5.4. Алгоритми голосування

Багато розподілених алгоритмів вимагають, щоб один із процесів був координатором, ініціатором або виконував іншу спеціальну роль. Зазвичай не важливо, який саме процес виконує ці спеціальні функції, головне, щоб такий процес взагалі існував. Розглянемо алгоритми, призначені для вибору координатора.

Якщо всі процеси абсолютно однакові, то способу обрати один з них не має, тому вважають, що кожен процес має **унікальний номер**, наприклад мережну адресу (для простоти вважатимемо, що на кожній машині виконується один процес). Переважно алгоритми голосування мають знаходити процес із максимальним номером і призначати його координатором. Алгоритми розрізняють за способами пошуку. За такими алгоритмами кожен процес знає номери всіх інших процесів, але не знає, які з них у цей час працюють, а які ні. Алгоритм голосування має гарантувати, що якщо голосування почалося, то воно, розглянувши всі процеси, вирішить, хто стане новим координатором.

5.4.1. Алгоритм «забіяки»

Як перший приклад розглянемо *алгоритм «забіяки» (bully algorithm)*. Коли один із процесів повідомляє, що координатор більше не відповідає на запити, то цей процес ініціює голосування. Процес, наприклад *P*, проводить голосування в такий спосіб: *P* надсилає всім процесам з більшими, ніж у нього, номерами повідомлення «*ГОЛОСУВАННЯ*».

Є два варіанти розвитку подій:

– якщо ніхто не відповідає, то *P* виграє голосування й стає координатором;

– якщо один із процесів з більшими номерами відповідає, то він стає координатором, а робота *P* на цьому закінчується.

У будь-який момент процес може одержати повідомлення «*ГОЛОСУВАННЯ*» від одного зі своїх колег з меншим номером, після чого одержувач надсилає відправникові повідомлення «*ОК*», показуючи, що він працює й готовий стати координатором. Потім одержувач сам ініціює голо-

сування. Зрештою, всі процеси, крім одного, не відповідатимуть, цей останній і буде новим координатором, який повідомить про свою перемогу відправленням усім процесам повідомлення, що він новий координатор і розпочинає роботу.

Процес, який перебував у неробочому стані, починаючи працювати, ініціює голосування. Якщо він виявляється процесом з найбільшим із працюючих процесів номером, то він виграє голосування й перебирає на себе функції координатора.

Приклад роботи алгоритму забіяки подано на рис. 6.8, на якому група складається з восьми процесів, пронумерованих від *0* до *7*. Раніше координатором був процес *7*, але він завис. Процес *4* першим побачив це й надсилає повідомлення «*ГОЛОСУВАННЯ*» всім процесам з номерами, більшими, ніж у нього, тобто процесам *5*, *6* і *7* (рис. 6.8, *а*). Процеси *5* і *6* відповідають (рис. 6.8, *б*). Після одержання першої з цих відповідей процес *4* розуміє, що він не стане координатором і координатором буде один з процесів з більшим номером. Процес *4* закінчує роботу й очікує, хто стане переможцем.

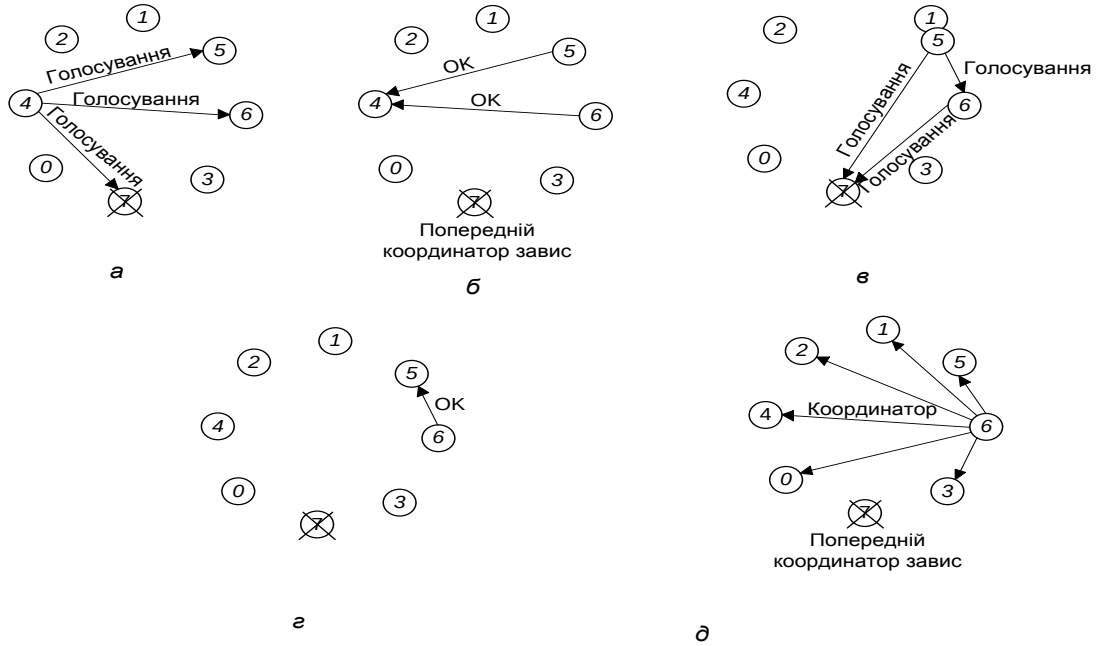


Рис. 6.8. Голосування за алгоритмом забіяки: *а)* – координатор, процес *7*, завис і процес *4* розпочав голосування; *б)* – відповіли процеси *5* і *6*; *в)* – процес *7* не відповідає; *г)* – процес *6* перебирає ініціативу на себе; *д)* – процес *6* стає координатором

Обидва процеси, які залишилися, *5* і *6*, продовжують голосування (рис. 6.8, *в*) і кожний надсилає повідомлення лише тим процесам, номери яких більші від їх власних. Процес *6* повідомляє процесу *5*, що голосування продовжуватиме він (рис. 6.8, *г*). У цей час процес *6* розуміє, що процес *7* «мертвий», а отже, переможець – він сам. Якщо інформація про стан зберігається на диску або іншому носії, коли з колишнім координатором що-небудь трапляється, то процес *6* має записати все потрібне. Готовий стати координатором процес *6* повідомляє про це розсиланням повідомлення «*КООРДИНАТОР»* усім працюючим процесам. Коли процес *4* одержить це повідомлення, він продовжить роботу від тієї операції, яку намагався виконати, коли виявив, що процес *7* «мертвий», використовуючи тепер як координатор процес *6*. Таким чином, уникнуто збою у процесі *7*, і робота системи триває.

Якщо процес *7* запуститься знову, то йому достатньо буде надіслати всім іншим процесам повідомлення «*КООРДИНАТОР»* і змусити їх працювати під його керівництвом.

5.4.2. Кільцевий алгоритм

Кільцевий алгоритм голосування грунтується на використанні кільця. На відміну від деяких інших кільцевих алгоритмів у цьому алгоритмі не потрібен маркер. Припускаємо, що процеси фізично або логічно впорядковані так, що кожний з процесів знає, хто його спадкоємець. Коли один із процесів виявляє, що координатор не функціонує, він формує повідомлення «*ГОЛОСУВАННЯ»*, яке містить його номер процесу, і надсилає його своєму спадкоємцеві. Якщо спадкоємець не працює, то відправник пропускає його й переходить до наступного елемента кільця або до наступного процесу, поки не знайде працюючий процес. На кожному кроці відправник додає свій номер процесу до списку в повідомленні, активно просуваючи себе як кандидата у координатори (рис. 6.9).

Зрештою, повідомлення повернеться до процесу, який розпочав голосування. Процес розпізнає цю подію за надходженням повідомлення, яке містить номер процесу. В цей момент тип повідомлення змінюється на

«*КООРДИНАТОР*» і знову відправляється по колу, цього разу щоб повідомити всім процесам, хто став координатором (елемент списку з максимальним номером) і які процеси входять у нове кільце. Після того, як це повідомлення один раз обійде все кільце, воно видаляється і процеси кільця повертаються до звичайної роботи.

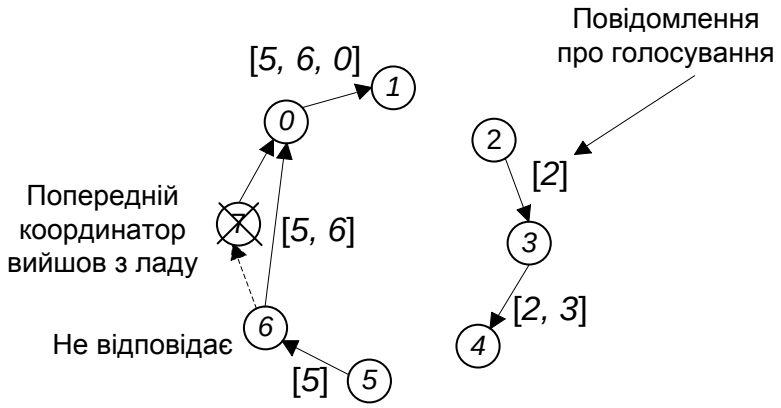


Рис. 6.9. Кільцевий алгоритм голосування

Якщо два процеси *2* і *5* одночасно виявляють, що попередній координатор, процес *7*, перестав працювати, то кожний з них формує повідомлення «*ГОЛОСУВАННЯ*» запускає його кільцем незалежно від другого. Зрештою, обидва повідомлення проходять усе кільце, та процеси *2* і *5* перетворюють їх у повідомлення «*КООРДИНАТОР*», які пересилаються тим самим елементам кільця у такому самому порядку. Коли ці повідомлення, у свою чергу, роблять повне коло, вони видаляються. Відпрацювання такого алгоритму в найгіршому випадку незначно завантажує мережу.

5.5. Взаємне виключення

Системи, що складаються з великої кількості процесів, зазвичай найпростіше програмувати, використовуючи **критичні області**. Коли процесу потрібно зчитати або обновити спільно використовувані структури даних, він спочатку входить у критичну область, щоб за допомогою взаємного виключення переконатися, що жоден із процесів не використає одночасно з ним

загальні структури даних. Розглянемо кілька способів реалізації критичних областей і взаємних виключень у розподілених системах.

5.5.1. Централізований алгоритм

Найбільш простий спосіб організації взаємних виключень у розподілених системах полягає в тому, щоб використовувати методи їх реалізації, обрані в однопроцесорних системах. Один із процесів обирають координатором (наприклад, процес *1*, запущений на машині з найбільшою мережною адресою). Щоразу, коли цей процес має ввійти у критичну область, він надсилає координаторові повідомлення із запитом, у якому повідомляє, у яку критичну область він входитиме, і запитує **дозвіл** на це. Якщо жоден з процесів у цей момент не перебуває в цій критичній області, то координатор надсилає відповідь з **дозволом на доступ**, як показано на рис. 6.10, *а*. Після одержання відповіді процес, який запросив доступ, входить у критичну область.

Припустімо, інший процес *2* запитує дозвіл на вхід у ту саму область (рис. 6.10, *б*), а координатор знає, що в ній уже перебуває інший процес і не дає дозволу на вхід. Конкретний спосіб заборони доступу залежить від особливостей системи. Координатор не відповідає, блокуючи цим процес *2*, який очікує відповіді (рис. 6.10, *б*), або може надіслати відповідь *« доступ заборонено »*. Під час кожного звертання координатор ставить запит від процесу *2* у чергу й очікує подальших повідомлень.

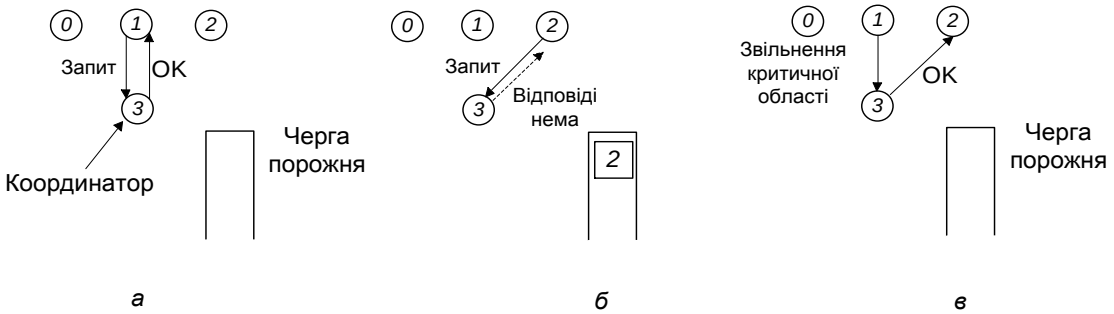


Рис. 6.10. Процес *1* запитує в координатора дозвіл на вхід у критичну область і отримує дозвіл (*а*); процес *2* запитує дозвіл на вхід у ту саму критичну область, але координатор не відповідає (*б*); процес *1* виходить із критичної області, він сповіщає про це координатора, який дає доступ процесу *2* (*в*)

Коли процес *1* залишає критичну область, він надсилає координаторові повідомлення, відмовляючись від ексклюзивного доступу на область, як показано на рис. 6.10, *в*. Координатор обирає перший елемент із черги відкладених запитів і надсилає процесу *2* повідомлення про дозвіл на доступ до критичної області. Якщо процес було блоковано (тобто для нього це перше повідомлення від координатора), то він розблокується і входить у критичну область; якщо процесу було відіслано повідомлення із забороною на доступ, то він опитуватиме повідомлення або блоки, які надходять, а потім побачивши дозвіл, увійде у критичну область.

Алгоритм гарантує взаємне виключення: координатор дозволяє ввійти в кожну критичну область одному процесу за один раз. Схема також проста в реалізації й застосовує для роботи з критичною областю лише три повідомлення *(запросити, дозволити й звільнити)*. Такий алгоритм може застосовуватися і для іншого виділення ресурсів, а не тільки для роботи з критичними областями.

5.5.2. Розподілений алгоритм

Наявність навіть одного непрацюючого місця в централізованих алгоритмах неприпустима, тому було запропоновано **розподілені алгоритми взаємного виключення** [12].

Централізований алгоритм вимагає повної впорядкованості подій у системі, тобто в будь-якій парі подій, наприклад у відправленні повідомлень, має бути відомо, яка з них відбулася першою. Алгоритм Лампорта є одним зі способів введення такої впорядкованості й може бути використаний для розміщення оцінок часу розподілених взаємних виключень.

Коли процес хоче ввійти у критичну область, він створює повідомлення, що містить ім’я критичної області, свій номер і поточний час, та відсилає це повідомлення всім процесам, концептуально включаючи самого себе. Передбачається, що відправлення повідомлення є надійним, оскільки на кожен лист надходить підтвердження щодо одержання. Замість окремих повідомлень може бути використаний доступний надійний груповий зв’язок.

Коли процес одержує повідомлення із запитом від іншого процесу, його подальші дії залежать від його зв'язку з тією критичною областю, ім'я якої зазначено в повідомленні. Можна назвати три варіанти виконання подальших дій процесом.

1. Якщо одержувач не перебуває у критичній області й не хоче туди входити, то він відсилає відправникові повідомлення «ОК».

2. Якщо одержувач перебуває у критичній області, то він не відповідає, а поміщає запит у чергу.

3. Якщо одержувач хоче ввійти у критичну область, але ще не зробив цього, то він порівнює відмітку часу повідомлення, яке надійшло, з відміткою часу повідомлення, яке він відіслав. Якщо повідомлення, яке надійшло, має менший номер, то одержувач відповідає відправленням повідомлення «ОК». Якщо його власне повідомлення має меншу відмітку часу, то одержувач ставить повідомлення, які надійшли, **у чергу**, нічого не надсилаючи.

Після відправлення повідомлення-запиту на доступ у критичну область процес припиняється й очікує, коли хто-небудь дасть йому дозвіл на доступ, і після того, як усі дозволи отримано, він може ввійти у критичну область. Коли процес залишає критичну область, він відсилає повідомлення «ОК» усім процесам, які містяться у черзі, й видаляє всі такі повідомлення зі своєї черги.

Розглянемо як працює алгоритм. Якщо конфліктів немає, то система працює стабільно, але якщо два процеси намагаються одночасно ввійти в одну критичну область, то виникає конфлікт (рис. 6.11, *а*).

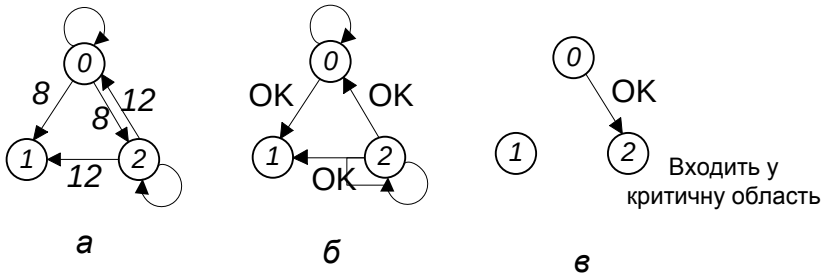


Рис. 6.11. Два процеси одночасно хочуть одержати доступ до однієї критичної області (*а*); процес *0* має меншу відмітку часу й тому виграє (*б*); коли процес *0* завершує роботу з критичною областю, він відправляє повідомлення «ОК», і процес *2* може ввійти у критичну область (*в*)

Процес *0* розсилає всім запит з відміткою часу *8*, а одночасно з ним процес *2* – запит з відміткою часу *12*. Процес *1* не збирається входити у критичну область і у відповідь надсилає повідомлення «ОК» їм обом. Процеси *0* і *2* помічають конфлікт, тому порівнюють відмітки часу. Процес *2* бачить, що програв, і дозволяє доступ процесу *0*, надсилаючи йому повідомлення «ОК». Процес *0* ставить відповідь від процесу *2* у чергу для подальшої обробки й входить у критичну область, як показано на рис. 6.11, *б*. Коли процес *0* закінчує роботу в критичній області, він видаляє відповідь *2* із черги повідомлень і відправляє процесу *2* повідомлення «ОК», дозволяючи йому ввійти у критичну область, що й показано на рис. 6.11, *в*. Алгоритм працює в усіх конфліктних ситуаціях, оскільки у разі конфліктів завжди виграє найменша відмітка часу.

Відзначимо, що показана на рис. 6.11 ситуація могла б докорінно змінитися, якби процес *2* надіслав своє повідомлення раніше, ніж процес *0*, та одержав дозвіл на доступ до створення своєї відповіді на повідомлення від процесу *0*. У цьому разі процес *2*, знаючи про те, що в момент відсилання відповіді він перебуває у критичній області, помістив би запит від процесу *0* у чергу, не надсилаючи ніякої відповіді.

Як і централізований алгоритм, алгоритм розподіленого взаємного виключення запобігає виникненню *тупиків* і *зависань*. Кількість повідомлень , які звертаються до одного процесу, $k = (-2\times(n-1))$, де n – загальна кількість процесів у системі. Більше немає єдиної точки, збій у якій міг би спричинити непрацездатність усієї системи.

Одна точка збою змінилася на n точок збою. Якщо який-небудь із процесів перестане працювати, то він не зможе відповісти на запит, що буде сприйнято як відмова в доступі й заблоковано подальші спроби всіх процесів увійти в яку-небудь з критичних областей. Імовірність того, що стане непрацездатним один з n процесів у n разів більша, ніж імовірність збою єдиного координатора, що є першим недоліком такого алгоритму.

Цей алгоритм може бути вдосконалено за допомогою такого прийому: коли надходить запит, його одержувач надсилає відповідь завжди, дозволяючи або забороняючи доступ. Щоразу, коли запит або відповідь втрачено, від-

правник чекає необхідний час і одержує відповідь, або вважає, що одержувач перебуває в неробочому стані. Після одержання заборони відправник чекає наступного повідомлення «ОК».

Другий недолік цього алгоритму полягає в тому, що або процеси, які ініціюють виклик, мають використовувати примітиви групового зв'язку, або кожен процес підтримує список групи самостійно, забезпечуючи внесення процесів у групу, видалення процесів з групи й відстеження збоїв. Метод якнайкраще працює тоді, коли група процесів мала, а членство у групі постійне й ніколи не змінюється.

Одним з недоліків централізованого алгоритму було проведення обробки всіх запитів у одному місці, що могло призвести до зниження продуктивності, а у розподіленому алгоритмі всі процеси змушені брати участь у всіх рішеннях, що стосуються входу в критичні області. Якщо один з процесів виявляється нездатним впоратися з таким навантаженням, то малоймовірно, що інші процеси зможуть зробити те ж саме паралельно.

У цей алгоритм внесено додаткові вдосконалення, у такому разі важливою умовою є запобігання одночасному входу двох процесів у критичну область. Алгоритм модифікується так, щоб дозволити процесу вхід у критичну область після того, як він збере дозволи більшості, а не всіх інших процесів. У цьому разі, після того як процес дасть дозвіл на вхід у критичну область одному з процесів, він не зможе дати такий самий дозвіл іншому процесу доти, доки перший не залишить критичну область. Можливі також й інші вдосконалення, але вони значно ускладнюють алгоритм.

5.5.3. Алгоритм маркерного кільця

Зовсім інший підхід до реалізації взаємних виключень у розподілених системах пропонує **алгоритм маркерного кільця** (Token Ring). Нехай наявна магістральна мережа (наприклад, Ethernet), але без внутрішнього впорядкування процесів (рис. 6.12, *а*). Програмно створюється логічне кільце, у якому кожному процесу призначається місце в кільці, як показано на рис. 6.12, *б,* за порядком проходження мережних адрес або якось інакше. Не важливо, як

саме задано впорядкованість, головне – як дати процесу знати, який процес у кільці є наступним після нього.

Під час ініціалізації кільця процес *0* одержує **маркер**, або токен (token), що циркулює кільцем, тобто передається від процесу *k* до процесів *k+1* (це модуль розміру кільця) наскрізними повідомленнями. Коли процес одержує маркер від свого «сусіда», він перевіряє, чи не потрібно йому ввійти у критичну область. Якщо це так, то він входить у критичну область, виконує там усю необхідну роботу й залишає область, після чого передає маркер далі. Входити в іншу критичну область, використовуючи той самий маркер, заборонено.

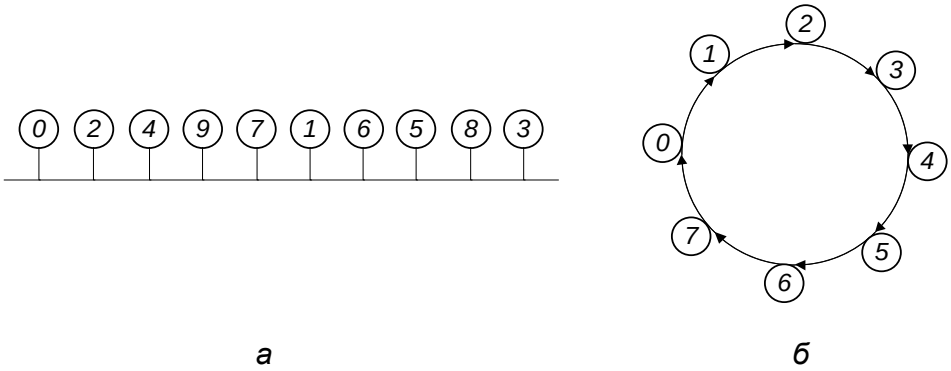


Рис. 6.12. Хаотична група процесів у мережі *(а)*; логічне кільце, створене програмно *(б)*

Якщо процес, одержавши від «сусідів» маркер, не зацікавлений у вході у критичну область, він передає цей маркер далі. Відповідно, якщо жоден процес не перебуває у критичних областях, то маркер циркулює кільцем з максимально можливою швидкістю.

Тільки один процес у будь-який момент має маркер, тому лише один процес може перебувати у критичній області. Оскільки маркер переміщується від процесу до процесу в загальновідомому порядку, зависання не відбувається. Коли процес вирішує ввійти у критичну область, йому доведеться чекати, поки всі інші процеси послідовно не ввійдуть у критичну область і не вийдуть із неї.

Цей алгоритм також має недоліки. Якщо маркер один раз втрачений, то він має бути відновлений. Зрозуміти, що він зник, достатньо складно, оскільки час між послідовними появами маркера в мережі не обмежений. Той факт,

що маркера не було видно протягом години, зовсім не означає, що його загублено, адже хто-небудь його може використовувати в цей час.

5.5.4. Порівняльна характеристика алгоритмів

Централізований алгоритм найбільш простий і ефективний, оскільки щоб увійти у критичну область, йому достатньо трьох повідомлень – запиту, дозволу на вхід і повідомлення про вихід. **Розподіленому алгоритму** (використовуються лише наскрізні комунікації) потрібно для запиту $n - 1$ повідомлень, по одному на кожен процес, і додатково $n - 1$ повідомлень на дозвіл, разом $2\times(n - 1)$. В **алгоритмі маркерного кільця** кількість повідомлень різна. Якщо кожний з процесів буде постійно вимагати входу в критичну область, кожне передавання маркера стане результатом виходу з критичної області одного з процесів і для входу в критичну область знадобиться одне повідомлення. В іншому, крайньому випадку маркер може циркулювати кільцем годинами, і ним ніхто не зацікавиться, тоді кількість повідомлень на вхід у критичну область одного процесу не обмежена.

Затримка з моменту, коли процесу знадобилося ввійти у критичну область, до моменту входу для цих трьох алгоритмів також різна. Якщо критичні області малі й використовуються рідко, то основним фактором затримки є механізм входу в критичну область. Якщо ж критичні області використовуються постійно, то основним фактором затримки є очікування свого входу. За централізованим алгоритмом, щоб увійти у критичну область потрібно лише два повідомлення; за розподіленим алгоритмом – $2\times (n - 1)$ повідомлень, з урахуванням того, що їх надсилають одне за другим; для алгоритму маркерного кільця час варіюється від *0* (маркер процесу, який входить у критичну область) до $n - 1$ (маркер був щойно переданий далі по кільцю).

Всі три алгоритми важко реагують на збої, тому щоб збій не призвів до повної непрацездатності системи, доводиться вживати спеціальних заходів і додатково ускладнювати алгоритм. Розподілені алгоритми більш чутливі до збоїв, ніж централізований. У захищених від збою системах неможливо застосувати жоден з розподілених алгоритмів, але якщо збої нечасті, то вони будуть працювати.

5.6. Розподілені транзакції

Концепція транзакцій тісно пов'язана з концепцією взаємних виключень, алгоритми яких забезпечують одночасний доступ не більш, ніж одного процесу до спільно використовуваних ресурсів, зокрема до файлів, принтерів . Транзакції теж захищають загальні ресурси від одночасного доступу декількох паралельних процесів, але можуть виконувати і багато інших функцій, наприклад, перетворювати процеси доступу й модифікації множини елементів даних в одну атомарну операцію. Якщо процес під час транзакції вирішує зупинитися на половині шляху й повернути назад, то всі дані відновлюються з тими значеннями й у тому стані, у якому вони були до початку транзакції.

5.6.1. Модель транзакцій

Комп'ютерна модель працює таким чином. Один процес повідомляє, що хоче почати транзакцію з одним або декількома іншими процесами, після чого вони можуть погоджувати різні умови, створювати й видаляти сутності, виконувати операції. Потім **ініціатор** пропонує всім підтвердити, що роботу виконано. Якщо всі це підтверджують, то результати затверджуються і стають постійними. Якщо один або кілька процесів відмовляються (чи до погодження в них виникають збої), то всі процеси повертаються до того стану, в якому були до початку транзакції, з усіма наслідками, що випливають із цього, для файлів, баз даних тощо. Така властивість **«всі або ніхто»** істотно спрощує алгоритми керування паралельним доступом.

Використання транзакцій у комп'ютерних системах бере початок із шістдесятих років двадцятого сторіччя, оскільки раніше дисків і мережних баз даних не було, а всі дані зберігалися на магнітних стрічках.

Приклад. Уявімо супермаркет з автоматизованою системою інвентаризації, у якому щодня після закриття комп'ютер починав роботу з двома магнітними стрічками. Перша з них містила повні відомості про товари на момент відкриття зранку поточного дня, а друга – список змін за день: продукти, які придбали покупці, та продукти, які повернуто пос-

тачальникам. Комп'ютер зчитував інформацію з обох стрічок та створював нову основну стрічку інвентаризації, як показано на рис. 6.13.

Основна перевага цієї схеми полягала в тому, що у разі будь-яких збоїв усі стрічки можна було перемотати на початок і почати роботу знову без жодних проблем. Такі примітивні старі магнітні стрічки мали властивість, характерну для транзакції, – «все або ніхто».

Приклад. Розглянемо сучасне банківське прикладне програмне забезпечення, яке вносить зміни в мережну базу даних. Клієнт дзвонить у банк, використовуючи комп'ютер з модемом, маючи намір зняти гроші з одного рахунку й покласти їх на другий. Операція здійснюється у два прийоми:

1. Зняти суму *a* з рахунку 1.
2. Покласти суму *a* на рахунок 2.

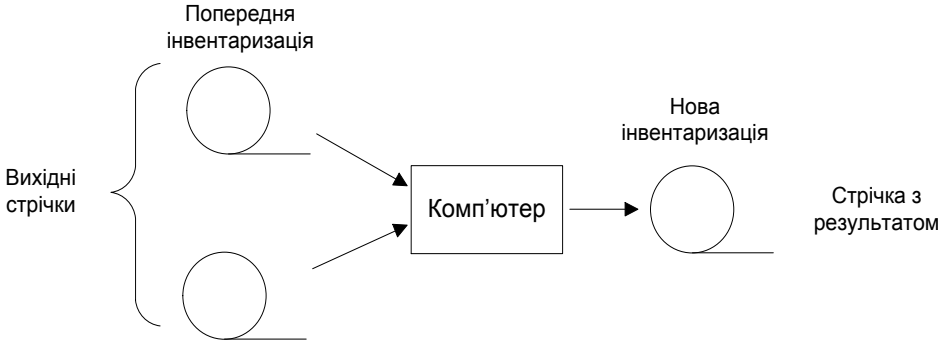


Рис. 6.13. Внесення змін в основну стрічку захищено від збоїв

Якщо модемне з'єднання після виконання першого етапу, але до початку виконання другого етапу буде розірвано, то гроші з першого рахунку будуть зняті, а на другий не надійдуть, тому для розв'язання цієї проблеми слід об'єднати ці операції в одну транзакцію: або обидві вони будуть виконані, або не буде виконана жодна. Отже, **ключовою** є можливість **відкату до вихідного стану** якщо **неможливо завершити транзакцію**. У такому разі дійсно потрібно мати спосіб «відмотати до початку» базу даних. Цю можливість дає транзакція.

Програмування із застосуванням *транзакцій* вимагає спеціальних примітивів, які можуть підтримуватися як базовою розподіленою системою, так і виконуючою системою мови програмування, типові приклади яких наведено в табл. 6.1. Повний список примітивів залежить від того, які об'єкти використовуються у транзакції: у поштовій системі примітивами можуть бути відправлення, прийом і пересилання пошти; у банківських системах – команди READ і WRITE, звичайні інструкції, виклики процедур й інші.

Таблиця 6.1. Деякі примітиви, які використовуються у транзакціях

Примітив	**Опис**
BEGIN_TRANSACTION	Позначити початок транзакції
END_TRANSACTION	Припинити транзакцію і спробувати завершити її
ABORT_TRANSACTION	Перервати транзакцію і відновити попередні значення
READ	Прочитати дані з файлу, таблиці або іншого джерела
WRITE	Записати дані у файл, таблицю або інший приймач

Для обмеження області дії транзакції використовуються примітиви *BEGIN_TRANSACTION* й *END_TRANSACTION*. Операції, що містяться між ними, та формують **тіло транзакції**, які можуть бути системними викликами, бібліотечними процедурами або інструкціями мовою реалізації.

Транзакції мають такі властивості:

атомарність *(atomic*) – для кінцевих користувачів транзакція неподільна;

несуперечливість *(consistent)* – транзакція не порушує інваріантів системи;

ізольованість *(isolated)* – транзакції, які виконуються одночасно, не впливають одна на одну;

довговічність *(durable)* – після завершення транзакції внесені нею зміни стають постійними.

На ці властивості часто посилаються за їх першими буквами – ACID.

Перша ключова властивість, що проявляється в усіх транзакціях, – **атомарність** – гарантує, що будь-яка транзакція або повністю виконується як одна неподільна одночасна операція, або зовсім не виконується. Поки транзакція перебуває у процесі виконання, інші процеси (незалежно від того, залучені вони у транзакцію чи ні) не можуть спостерігати жодного проміжного стану.

Приклад. Уявімо, що транзакція починається, щоб додати дані в якийсь файл із початковою довжиною 10 байт. Якщо цей файл у процесі виконання транзакції побажає прочитати інший процес, то він побачить лише вихідні 10 байт незалежно від того, скільки байтів було додано у файл. Якщо транзакція завершиться успішно, то файл миттєво збільшиться до нового розміру, якщо ні – то до розміру на момент завершення без проміжних станів незалежно від того, скільки операцій усередині транзакції зумовило його збільшення.

Друга властивість – це **несуперечність -** означає, що якщо в системі до початку транзакції були якісь інваріанти, які вона постійно має зберігати, то вони будуть зберігатися й після її завершення.

Приклад. Так, у банківській системі ключовим інваріантом є закон збереження грошей. Після будь-яких внутрішніх переказів обсяг коштів у банку має зберігатися таким самим, як до переказів, хоча на короткий період у процесі виконання транзакції цей інваріант може порушуватися, але таке порушення не буде помітним ззовні.

Третя властивість – це **ізольованість** або **серіалізованість -** тобто якщо дві або більше транзакції відбуваються одночасно, то для кожної з них і для інших процесів підсумковий результат виглядає так само, як коли всі транзакції виконуються послідовно в якомусь, незалежному від системи, порядку.

Четверта властивість – це **довговічність** – тобто після завершення транзакцій подальші дії не мають жодного значення, адже транзакція закінчена, а її результати незмінні. Ніякі збої після завершення транзакції не можуть призвести до знищення результатів транзакції або їх втрати.

5.6.2. Класифікація транзакцій

Транзакція – серія операцій, що задовольняє властивостям ACID. Цей тип транзакції називають також **плоскою транзакцією** *(flat transaction),* найбільш простим і поширеним типом транзакцій. Розрізняють два важливі класи транзакцій: **вкладені** й **розподілені** транзакції.

Плоскі транзакції мають безліч обмежень, які змушують шукати альтернативні моделі. Основне обмеження плоских транзакцій полягає в тому, що вони не можуть давати часткового результату в разі завершення або переривання. Інакше кажучи, сила атомарності плоских транзакцій є в той же час і їх слабкістю.

Приклад. Розглянемо web-сайт, гіперпосилання в якому двонапрямлені, тобто якщо web-сторінка *W1* містить *URL* сторінки *W2*, то *W1* знає, що *W2* посилається на неї. У разі коли сторінка *W2* перенесена в інше місце або замінена іншою, то всі гіперпосилання на *W2* мають бути оновленими, причому бажано, щоб однією атомарною операцією, інакше можемо загубити посилання на *W2*. Теоретично за таких умов можна використати плоску транзакцію, яка складається з внесення змін у *W2* і серії операцій, кожна з яких змінює одну web-сторінку, яка містить гіперпосилання на *W2*.

Проблема полягає в тому, що така транзакція може вимагати для виконання декількох годин. Сторінки, які посилаються на *W2*, можуть бути розкидані по всьому Internet, а

таких сторінок, які потребують виправлення, може бути тисячі. Робити виправлення окремими транзакціями не є ефективним тому, що деякі web-сторінки міститимуть правильні посилання, а деякі ні. Можливим рішенням у цьому разі було б завершення змін зі збереженням старого посилання на *W2* для тих сторінок, посилання на які ще не змінилися.

У зв'язку з тим, що дані та обчислення стають все більш розподіленими, атомарність плоских транзакцій стає істотно незручною. Згідно з правилами обробки плоских транзакцій усі компоненти глобальної транзакції мають успішно завершитися або не завершується жодна з них. Якщо невдало закінчилася тільки зміна однієї віддаленої бази даних, керована деяким менеджером ресурсів, то і всі інші компоненти повернутися до стану, який передував початку транзакції.

З огляду на кількість інформації, яка обробляється у великій або навіть середній організації з великою кількістю серверів LAN на персональних комп'ютерах і, можливо, з мобільними базами даних, можна припустити, що ймовірність відмови хоча б одного вузла достатньо висока. Якщо застосовується модель плоских транзакцій, то доведеться заново виконувати всі складові частини транзакції, що істотно підвищує вимоги до обчислювальних ресурсів і споживає значну частку пропускної здатності системи.

Очевидно, що під час розподілених обчислень необхідно якимось чином здійснювати декомпозицію плоских транзакцій, зокрема за допомогою використання **модифікації моделі плоских транзакцій**, яка зберігає атомарність, але знижує потребу в повторному виконанні дій (тобто в «переробках») та містить поняття «**контрольні точки».**

Контрольні точки встановлюються у прикладному програмному забезпеченні для того, щоб з'ясувати моменти, починаючи з яких можна продовжити обчислення в разі виникнення проблем. В ідеальному випадку контрольні точки мають відповідати частково узгодженим станам.

Після досягнення чергової контрольної точки у транзакції створюється нова атомарна дія, яка запускається на виконання. Тільки остання атомарна дія всієї послідовності може виконати **фіксацію** (*COMMIT WORK*) транзакції; оператор *COMMIT WORK* передається всім попереднім атомарним діям, поки всі вони не будуть зафіксовані. На відміну від моделі багатоланкових тран-

закцій, контрольна точка не призводить до незворотної фіксації, виконаної до цього моменту роботи.

Переривання (*ROLL BACK*), або **відкати**, транзакції можуть ініціюватися з будь-якої атомарної дії, крім останньої. Це означає, що якщо для якоїсь атомарної дії була досягнута контрольна точка, то для цієї дії вже не може бути в подальшому прийнято рішення про відкат. Відкат може бути виконаний до будь-якої з попередніх контрольних точок, тому менеджер обробки транзакцій повинен ураховувати параметр, який вказує, до якої саме контрольної точки потрібно зробити відкат (в ідеальному випадку логіка програми має передбачати визначення контрольної точки, до якої у разі невдачі слід відкотити виконання).

Варто зауважити, що проводилися дослідні роботи з вивчення доцільності стійких *(persistent)* контрольних точок, які фіксуються в дисковій або іншій довгостроковій пам'яті для того, щоб результати виконання були доступними після системних збоїв. Однак немає поки що ніякого рішення, яке можна було б прийняти беззастережно. Дві нові моделі транзакцій – **вкладені** й **багатоланкові** транзакції – надають схожі можливості при тому, що формальні визначення цих понять значно краще опрацьовані й загальноприйняті.

Розглянемо **модель багатоланкових транзакцій**, яка концептуально подібна моделі контрольних точок, але допускає фіксацію частини роботи, виконаної до деякого моменту, можливість відкату зафіксованих дій виключається.

У межах багатоланкової транзакції зберігаються всі необхідні елементи контексту виконання (курсори баз даних, відкриті файли тощо), хоча ресурси, які стали непотрібними, можна звільняти.

Модель багатоланкових транзакцій містить оператор *CHAIN WORK* – неподільну комбінацію операторів *COMMIT WORK* і *BEGIN WORK*, яка нерівноцінна послідовному виконанню операторів *COMMIT WORK* і *BEGIN WORK* окремо. У разі виконання цих операторів окремо контекст зникає; деяка інша транзакція може «вклинитися» і змінити значення в базі даних, потрібні для виконання наступної «ланки» багатоланкової транзакції раніше, ніж ця ланка почне виконуватися. Таким чином, багатоланкові транзакції концепту-

ально еквівалентні транзакціям з контрольними точками з тією відмінністю, що відкат може здійснюватися тільки до останньої зафіксованої точки, а не до будь-якої попередньої контрольної точки.

Обидві моделі транзакцій – багатоланкові та з контрольними точками – дозволяють описувати послідовність дій; відмінності стосуються лише можливостей відкату і стійкості дій, виконаних до заданої точки.

Деякі з тих обмежень можуть бути зняті у процесі використання **вкладених** транзакцій *(nested transactions).* **Транзакція верхнього рівня** може поділятися на **дочірні** транзакції, які працюють паралельно на різних машинах, для підвищення продуктивності або спрощення програмування. Кожна із цих дочірніх транзакцій також може складатися з однієї або більше транзакцій або, у свою чергу, поділятися на дочірні транзакції.

Коли починається будь-яка транзакція або вкладена транзакція, створюється закрита копія даних усієї системи. Якщо внутрішня транзакція переривається, то її внутрішній поточний стан зникає; якщо вона завершується, то її внутрішній поточний стан переміщується в батьківську транзакцію. Таким чином, якщо дочірня транзакція завершується й починається нова дочірня транзакція, то друга з них одержує можливість працювати з результатами, створеними першою. Аналогічно, якщо переривається об’ємна (верхнього рівня) транзакція, то всі вкладені в неї дочірні транзакції також перериваються.

Вкладені транзакції важливі для розподілених систем, тому що вони надають природний спосіб розподілу транзакцій по декількох машинах, зокрема розподіляють роботу вихідної транзакції між вкладеними підтранзакціями логічним способом.

Приклад. Вкладена транзакція резервування місця в літаку від Києва до Москви може потребувати доступу до двох баз даних, що належать кожному із цих міст. У цьому разі вкладена транзакція може поділятися на менші вкладені транзакції, яких логічно вже немає, оскільки резервування є неподільною операцією.

За умов, наведених у прикладі, вкладені транзакції (плоскі) працюють і *з* даними, розподіленими по декількох машинах. Такі транзакції називають **розподіленими** транзакціями (*distributed transactions*). Відмінність між вкладеними й розподіленими транзакціями невелика, але істотна. **Вкладені транза-**

кції – це транзакції, які логічно поділяються на ієрархічно організовані дочірні транзакції. Натомість **розподілені транзакції** логічно є плоскими неподільними транзакціями, які працюють із розподіленими даними. Цю відмінність ілюструє рис. 6.14.

Основна проблема розподілених транзакцій полягає у тому, що для блокування даних і підтвердження транзакції необхідні окремі розподілені алгоритми.

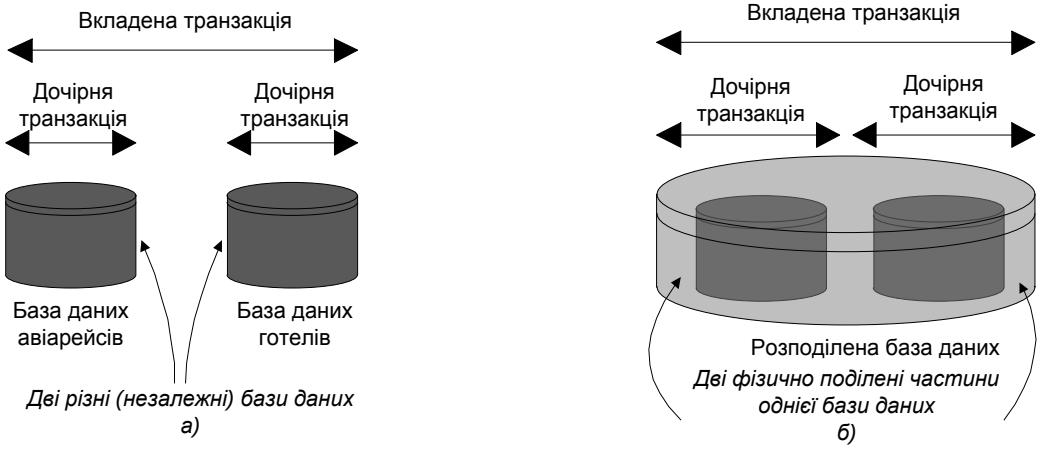


Рис. 6.14. Вкладена транзакція (*а*); розподілена транзакція (*б*)

5.6.3. Рівні ізольованості транзакцій

Рівень ізольованості транзакцій – значення, що визначає рівень, за якого у транзакції допускаються неузгоджені дані, тобто ступінь ізольованості однієї транзакції від другої. Більш високий рівень ізольованості підвищує точність даних, але при цьому може зменшуватися кількість паралельно виконуваних транзакцій, а більш низький рівень ізольованості дозволяє виконувати більше паралельних транзакцій, але знижує точність даних.

У разі паралельного виконання транзакцій можуть виникати такі проблеми:

- втрачені оновлення *(lost update)*;
- «брудне» читання *(dirty read)* - читання даних, записаних транзакцією, яку відкотили;
- неповторюване читання *(non-repeatable read);*
- фантомна вставка *(phantom insert).*

Розглянемо ситуації, в яких можливе виникнення таких проблем.

Приклад. Втрачене оновлення. Припустімо, є дві транзакції, відкриті різними прикладними програмами, в яких виконані такі SQL-оператори, як показано на на рис. 6.15, де видно, що у транзакції *1* змінюється значення поля *f2*, а потім у транзакції *2* також змінюється значення цього поля, у результаті чого зміну, виконану першою транзакцією, буде втрачено.

«Брудне» читання. Припустімо, є дві транзакції, відкриті різними прикладними програмами, в яких виконані такі SQL-оператори, які наведено на рис. 6.16 у транзакції *2* вибирається значення поля *f2*, потім у транзакції *1* змінюється значення цього поля. У разі повторної спроби вибрати значення з поля *f2* транзакція *2* отримає інший результат. Це є особливо неприйнятним, коли дані зчитуються з метою їх часткової зміни і зворотного запису в базу даних.

Транзакція 1	Транзакція 2
Момент 1: SELECT f2 FROM tbl1 WHERE f1=1;	Момент 1 : SELECT f2 FROM tbl1 WHERE f1=1;
Момент 2: UPDATE tbl1 SET f2=20 WHERE f1=1;	Момент 2: ----
Момент 3: ------	Момент 3: UPDATE tbl1 SET f2=25 WHERE f1=1;

Рис. 6.15. Суперечливі транзакції, що призводять до втраченого оновлення

Момент 1: COMMIT;	Момент 1: ------
Момент 2: -------	Момент 2: SELECT f2 FROM tbl1 WHERE f1=1;

Рис. 6.16. Приклад «брудного» читання

Фантомна вставка. Припустімо, є дві транзакції, відкриті різними прикладними програмами, в яких виконані такі SQL-оператори, які показано на рис. 6.17.

Транзакція 1	Транзакція 2
Момент 1: -----	Момент 1: SELECT SUM(f2) FROM tbl1;
Момент 2: INSERT INTO tbl1 (f1,f2) VALUES (15,20);	Момент 2: ----
Момент 3: -------	Момент 3: SELECT SUM(f2) FROM tbl1;

Рис. 6.17. Приклад фантомної вставки

У транзакції *2* виконується SQL-оператор, який використовує всі значення поля *f* , а у транзакції *1* виконується вставка нового рядка, яка призводить до того, що повторне виконання SQL-оператора у транзакції *2* видасть інший результат. Таку ситуацію називають фантомною вставкою, яка є окремим випадком неповторюваного читання, при чому, якщо виконується SQL-оператор, який вибирає не всі значення поля *f2*, а тільки значення одного рядка таблиці (використовується предикат WHERE), то виконання оператора INSERT не призведе до фантомної вставки.

В ідеалі транзакції різних користувачів мають виконуватися так, щоб створювалася ілюзія, що користувач поточної транзакції єдиний. Проте в реальності, з огляду на продуктивність і для виконання деяких спеціальних завдань, СУБД надають різні рівні ізоляції транзакцій, описані у міру збільшення ізоляції транзакцій і надійності роботи з даними.

0 – непідтверджені читання (*Read Uncommited, Dirty Read*, брудне читання) – читання незафіксованих змін своєї транзакції та конкуруючих транзакцій, можливі нечисті, неповторювані читання і фантоми. Цей рівень гарантує тільки фізичну цілісність даних: якщо кілька користувачів одночасно змінюють один рядок, то в кінцевому варіанті рядок матиме значення, присвоєне користувачем, який останнім змінив запис, а не змішані значення стовпців окремих користувачів, тобто відбувається пошкодження даних. Для транзакції не встановлюється ніякого блокування, яке б гарантувало цілісність даних.

1 – підтверджене читання (*Read Commited*) – читання всіх змін своєї транзакції та зафіксованих змін конкуруючих транзакцій, нечисті читання неможливі, можливі неповторювані читання і фантоми. Цей рівень прийнято за замовчуванням для Microsoft SQL Server. Завершене читання, за якого немає чорнового, «брудного» читання, тобто це читання одним користувачем даних, не зафіксованих у БД командою *COMMIT*. Проте у процесі роботи однієї транзакції друга може бути успішно завершена і внесені нею зміни зафіксовані, у результаті чого перша транзакція працюватиме з початковим набором даних та не врахує змін, які відбулися у процесі її роботи через функціонування другої транзакції. Цю проблему називають **неповторюваним читанням.** В Oracle блокування на читання немає, натомість «читаюча» транзакція отримує ту версію даних, яка була актуальною у базі до початку «пишучої».

2 – повторюване читання (*Repeatable Read, Snapshot*) – читання всіх змін своєї транзакції, будь-які зміни, внесені конкуруючими транзакціями після початку своєї транзакції недоступні, нечисті й неповторювані читання неможливі, можливі фантоми. Рівень, за якого читання одного рядка або рядків у транзакції дає однакові результати, тобто, поки транзакція не завершена, ніякі інші транзакції не можуть модифікувати ці дані.

3 – упорядковані транзакції (*Serializable*), це відповідає рівню ізольованості, за якого транзакції виконуються суворо послідовно одна після одної, тобто транзакції, результат дії яких не залежить від порядку виконання кроків транзакції (заборонено читання всіх даних, змінених від початку роботи транзакції, у тому числі своєю транзакцією). Фантоми неможливі. Вирізняється найвищим рівнем ізольованості, коли транзакції повністю ізольовані одна від одної. На цьому рівні вважається, що результати паралельного виконання транзакцій для бази даних здебільшого збігаються з послідовним виконанням таких самих транзакцій (по черзі в будь-якому порядку).

Чим вищий рівень ізоляції, тим більше потрібно ресурсів, щоб їх підтримувати. Рівні ізольованості й проблеми, які на них вирішуються наведено у табл. 6.2.

Таблиця 6.2. Рівні ізольованості й проблеми, які на них виникають

Рівень ізольованості	**Фантомна вставка**	**Неповторюване читання**	**«Брудне» читання**	**Втрачене оновлення**
3	Запобігає	Запобігає	Запобігає	Запобігає
2	Не запобігає	Запобігає	Запобігає	Запобігає
1	Не запобігає	Не запобігає	Запобігає	Запобігає
0	Не запобігає	Не запобігає	Не запобігає	Не запобігає

У СУБД рівень ізольованості транзакцій можна обрати як для всіх транзакцій відразу, так і для однієї конкретної транзакції. За стандартом бази даних використовують рівень 1 (*Read Commited*). Рівень 0 використовується переважно для відстеження змін тривалих транзакцій або для читання даних, які рідко змінюються, а рівні 2 і 3 у разі підвищених вимог до ізольованості транзакцій.

5.6.4. Реалізація розподілених транзакцій

Розглянемо транзакції, які працюють із файловою системою. Якщо кожен процес, який виконує транзакцію, оновлює файл там, де файл використовується, то транзакція не буде атомарною, тому в разі переривання транзакції всі зміни, які порушили цілісність даних файлу під час оновлення, не зникнуть. Для запобігання цьому зазвичай використовують два методи: закритий робочий простір та журнал з попереджувальним записом.

Закритий робочий простір. Коли процес починає транзакцію, він одержує закритий робочий простір, який містить усі файли, до яких він хоче одержати доступ. Поки транзакція не завершиться або не перерветься, всі операції читання й запису будуть відбуватися не у файловій системі, а в закритому робочому просторі, тому перший метод реалізації процесу обробки транзакції – метод, який передбачає створення для процесу в момент початку транзакції закритого робочого простору.

Недолік цього методу полягає в тому, що ціна копіювання всього обсягу інформації у закритий робочий простір надто велика, що є рівносильним забороні на копіювання, тому під час використання цього методу потрібно здійснювти **оптимізацію.**

В основу першого варіанта оптимізації покладено міркування, що якщо процес тільки читає файл, але не змінює його, то у створенні закритої копії немає потреби. Він може використати реальний об'єкт, але за умови, що він не змінюється у процесі функціонування транзакції. Коли процес починає транзакцію, він має створити закритий робочий простір, у якому нічого не зміниться, за винятком вказівника на робочий простір свого батька. Якщо це транзакція верхнього рівня, то робочим простором батька буде файлова система. Потім можливі дві ситуації: перша, коли процес відкриває файл на читання і виконує пошук за вказівниками, поки файл не з'явиться в робочому просторі батька (або більш далекого предка); друга, коли файл відкривається на запис, він може бути знайдений у такий самий спосіб, як і у разі читання за винятком того, що спочатку він буде скопійований у закритий робочий простір.

У разі другого варіанта оптимізації вдається відмовитися від більшої частини копіювання, тобто замість цілого файлу в закритий робочий простір можна копіювати лише **індекс** файлу. **Індекс** – це блок даних, асоційованих із кожним файлом, який зберігає інформацію про те, де на диску містяться його блоки. У UNIX індекс – це індексний вузол (*inode*). У разі використання внутрішнього індексу файл може бути зчитаний так само, оскільки диск, до якого відбувається адресація, – це той самий диск із вхідним розміщенням блоків. Коли блок файлу модифікується, створюється його копія і в індекс вставляється адреса копії, як показано на рис. 6.18, *а,* після цього блок можна змінювати, не впливаючи на оригінал. Блоки, які додаються, обробляються так само. Нові блоки іноді називають **тіньовими блоками** *(shadow blocks).*

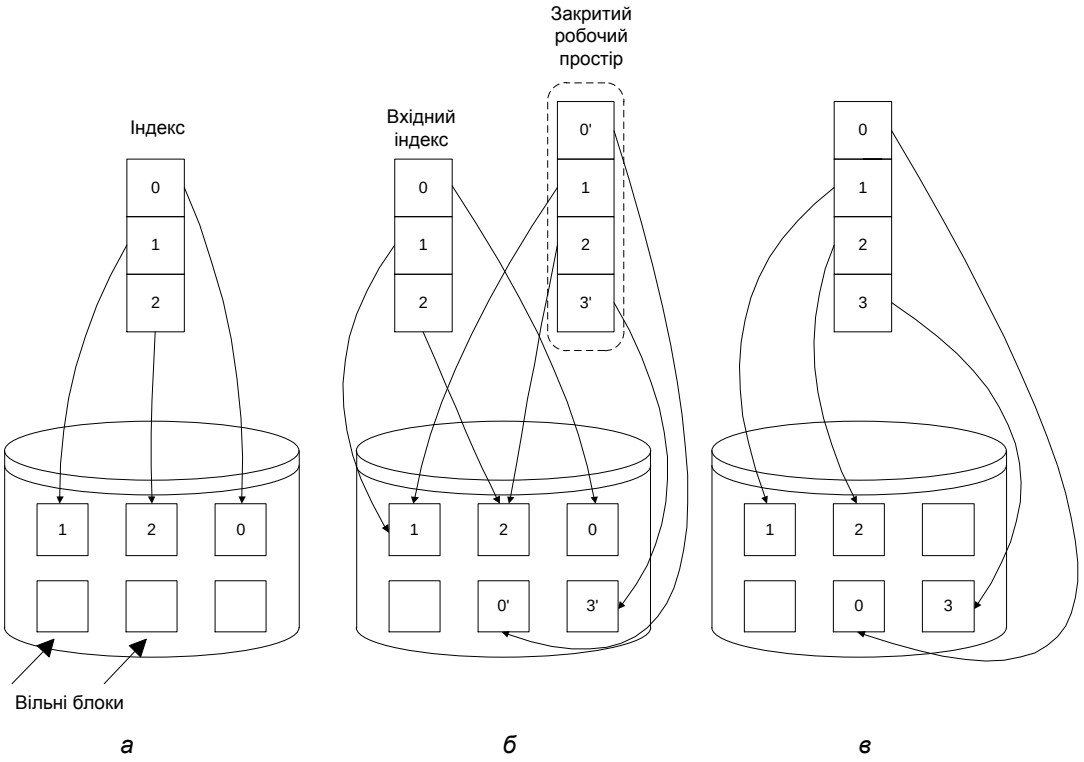


Рис. 6.18. Індекс файлу і дискові блоки для файлу з трьох блоків (*а*); ситуація після того, як транзакція модифікувала блок *0* і додала блок *3* (*б*); ситуація після підтвердження транзакції (*в*)

На рис. 6.18, *б* процес, який виконує транзакцію, бачить модифікований файл, а всі інші процеси продовжують працювати з початковим файлом. В особливо складних транзакціях закритий робочий простір може містити не один, а велику кількість файлів. Якщо транзакція переривається, то закритий робочий простір знищується і всі закриті блоки, які він містив, повертаються у список вільних. Якщо транзакція завершується, то закриті блоки у процесі атомарної операції переміщуються в робочий простір батька, як показано на рис. 6.18, *в*. Блоки, які більше не потрібні, повертаються у список вільних.

Ця схема працює також і для розподілених транзакцій, тоді аналогічний процес починається на кожній машині, яка містить файли, до яких у процесі виконання транзакції виконуватиметься доступ. Кожен процес одержує власний закритий робочий простір. Якщо транзакція переривається, то всі процеси відмовляються від своїх закритих робочих просторів. Натомість у разі підтвердження транзакції виконуються локальні оновлення, після чого транзакції можна вважати повністю завершеними.

Журнал з попереджувальним записом. Крім методу реалізації транзакції із закритим робочим простором використовують метод із журналом з попереджувальним записом (*write-ahead log*). Відповідно до цього файли модифікуються там, де і перебувають, але перед тим, як який-небудь блок дійсно буде змінений, у журнал заноситься запис із відомостями про те, яка транзакція вносить зміни, який файл і блок змінюються, які попередні й нові значення. Тільки після успішного запису в журнал зміни вносяться у файл.

Якщо транзакція успішна, то вона підтверджується й у журнал додається запис про підтвердження, але структури даних не змінюються, оскільки вони й так змінені. Якщо транзакція переривається, то журнал використовується для відновлення початкового стану. Починаючи з кінця і просуваючись до початку, із журналу зчитуються записи, й описані в них зміни скасовуються. Цю дію називають **відкатом** *(rollback)* **транзакції.**

Як і попередня, ця схема також може використовуватися для роботи з розподіленими транзакціями, коли кожна машина підтримує власний журнал змін у локальній файловій системі. Відкат у разі переривання транзакції кожна машина виконує окремо, відновлюючи свої вхідні файли.

5.6.5. Керування паралельним виконанням транзакцій

Мета керування паралельним виконанням транзакцій полягає в тому, щоб дозволити декільком транзакціям виконуватися одночасно, але таким чином, щоб набір оброблюваних елементів даних (наприклад, файлів або записів бази даних) залишався несуперечливим. **Несуперечність** досягається в результаті того, що доступ транзакцій до елементів даних організується в певному порядку так, щоб кінцевий результат був таким самим, як і під час виконання всіх транзакцій послідовно.

Керування паралельним виконанням найкраще можна зрозуміти в термінах трьох менеджерів, організованих за рівнями, як це показано на рис. 6.19. Нижній рівень надає **менеджера даних** (*data manager*), що фізично здійснює операції читання й запису даних. Менеджер даних нічого не знає про те, яка транзакція виконує читання або запис.

На середньому рівні міститься **планувальник** *(scheduler*), який несе основну відповідальність за правильність керування паралельною роботою, визначаючи, якій транзакції та у який момент дозволяється передати операцію читання або запису менеджерові даних, а також планує окремі операції читання й запису з гарантіями несуперечності й ізольованості.

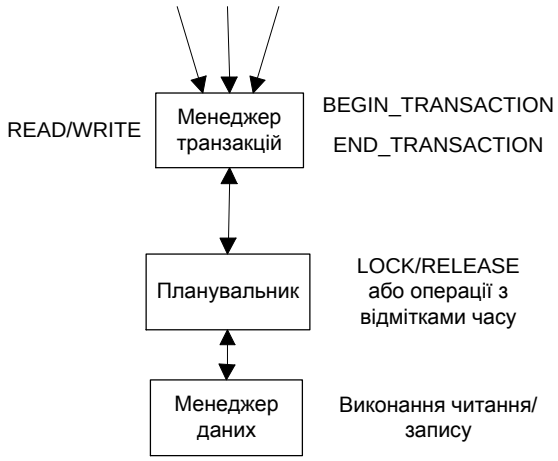


Рис. 6.19. Узагальнена організація менеджерів для керування транзакціями

На найвищому рівні перебуває **менеджер транзакцій**, який відповідає, насамперед, за атомарність і довговічність, й опрацьовує примітиви транзакцій, перетворюючи їх у запити до планувальника.

Модель, наведена на рис. 6.19, може бути адаптована до використання в розподілених системах, як показано на рис. 6.20, на якому кожна машина має своїх планувальника й менеджера даних, які спільно забезпечують гарантії несуперечності локальних даних. Кожна транзакція обробляється одним менеджером транзакцій, який працює з планувальниками окремих машин. Залежно від алгоритму керування паралельним виконанням транзакцій планувальник також може працювати з віддаленими менеджерами даних.

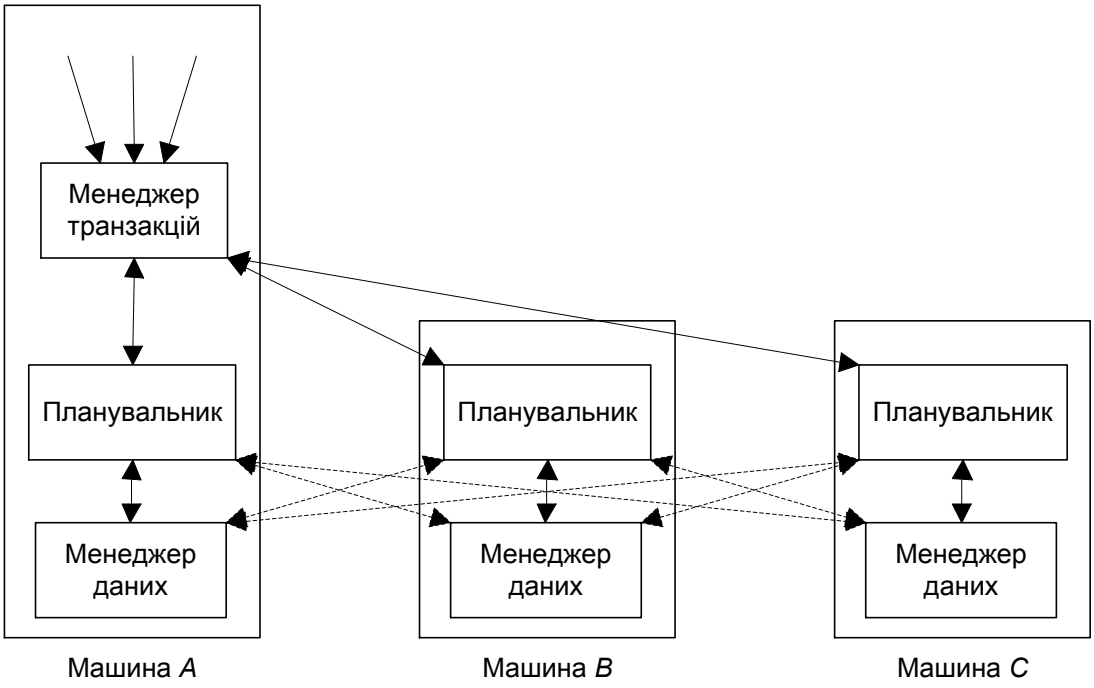


Рис. 6.20. Узагальнена організація менеджерів для керування розподіленими транзакціями

Ізольованість. Основним завданням алгоритмів керування паралельним виконанням є гарантія можливості одночасного виконання численних транзакцій доти, доки вони ізольовані одна від одної. Це означає, що підсумковий результат їх виконання буде таким самим, коли б ці транзакції виконувалися одна за другою в певному порядку. Різноманітні способи впорядкування називають **планами *(schedules).***

Основна ідея керування паралельним виконанням полягає в тому, щоб правильно спланувати операції, які конфліктують *(conflicting operations).* Дві операції **конфліктують,** якщо вони працюють з одним елементом даних, і як мінімум одна з них – це операція запису. В конфлікті читання-запису (*read-write conflict*) запис – це одна з операцій. Крім того, наявний конфлікт подвійного запису (*write – write conflict*), за якого немає значення, чи належать операції, що конфліктують, до однієї транзакції або до різних. Слід зазначити, що дві операції читання ніколи не конфліктують між собою.

Алгоритми керування паралельним виконанням класифікують за способом **синхронізації** операцій читання й запису, яка може виконуватися за допомо-

гою механізму взаємного виключення спільно використовуваних даних (тобто блокування) або явного впорядкування операцій за допомогою оцінок часу.

Подальший розподіл можна здійснити між **оптимістичним** і **песимістичним** керуванням за рахунок паралельного виконання. Для песимістичних підходів фундаментальним є закон Мерфі, відповідно до якого якщо щонебудь можна зробити неправильно, то воно буде зроблено неправильно, тобто операції синхронізуються до їх виконання, і таким чином конфлікти розв'язуються до того, як вони проявляться. Натомість **оптимістичні** підходи грунтуються на ідеї, що зазвичай нічого поганого не трапляється, тому операції без проблем виконуються, а синхронізація здійснюється в кінці транзакції. Якщо в цей момент з'являється конфлікт, то одна або більше транзакцій перериваються.

Двофазне блокування. Найбільш старий і поширений алгоритм керування паралельним виконанням транзакцій – це **блокування** (*locking*). У своїй найпростішій формі, коли процес під час виконання транзакції потребує читання або запису елемента даних, він просить планувальника заблокувати для нього цей елемент даних. Коли необхідність у цьому елементі даних зникає, процес просить планувальника зняти блокування. Завдання планувальника полягає в тому, щоб установлювати і знімати блокування таким чином, щоб одержувати працездатні плани виконання, тобто наявна потреба в застосуванні алгоритму, який надає лише серіалізовані плани, зокрема двофазного блокування.

У разі **двофазного блокування** (*Two-Phase Locking, 2PL*), продемонстрованого на рис. 6.21, планувальник спочатку на фазі підйому *(growing phase)* встановлює всі необхідні блокування, а потім на фазі спаду *(shrinking phase)* знімає їх.

У разі двофазного блокування виконуються такі три правила:

1. Коли планувальник одержує операцію *oper (T, x)* від менеджера транзакцій, він перевіряє, чи не конфліктує ця операція з іншими, які вже одержали блокування. Якщо наявний конфлікт, то операція *oper (T, x)* відкладається (і транзакція *T* також). Якщо конфлікту немає, то планувальник виконує блокування для елемента даних *x* і передає операцію менеджерові даних.

2. Планувальник ніколи не знімає блокування з елемента даних *x*, якщо менеджер даних повідомляє його, що він здійснює операцію, у якій бере участь цей елемент даних.

3. Після того як планувальник знімає з даних блокування, встановлене на вимогу транзакції *T*, він ніколи не робить на вимогу цієї транзакції інше блокування, при чому неважливо, на який елемент даних транзакція *T* вимагає встановити блокування. Будь-які спроби *T* встановити нове блокування є помилкою програмування й призводять до переривання транзакції *T*.

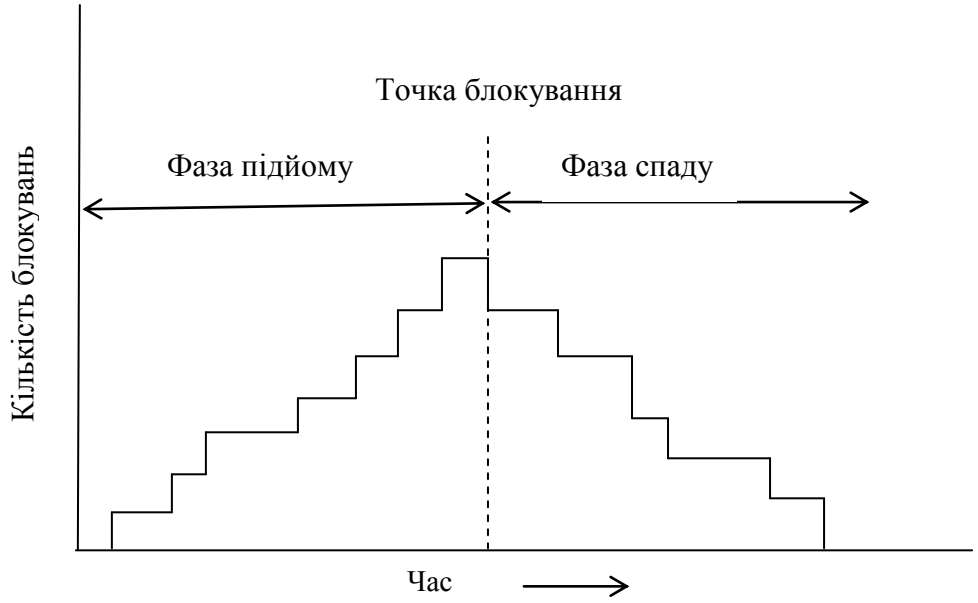


Рис. 6.21. Двофазне блокування

Якщо всі транзакції використовують двофазне блокування, то будь-який план, сформований перекриттям цих транзакцій, придатний до серіалізації, тому алгоритми двофазного блокування є поширеними.

У багатьох системах фаза спаду не починається доти, доки транзакція не закінчиться підтвердженням або перериванням, що і приведе до зняття блокувань, як показано на рис. 6.22. Такий режим, названий **строгим двофазним блокуванням** (*strict two-phase locking*), має дві суттєві переваги. По-перше, транзакція завжди зчитує значення, записані підтвердженою транзакцією, тому переривання транзакції неможливе через те, що її обчислення грунтуються на недоступних даних. По-друге, будь-які встановлення та зняття блокувань можуть виконуватися системою без використання транзакцій, тобто

будь-які блокування встановлюються у разі доступу до елемента даних і знімаються по закінченні транзакції. За такої поведінки усуваються **каскадні переривання** (*cascaded aborts*), які скасовують підтверджені транзакції з тієї причини, що одержано елемент даних, який одержувати не дозволялося.

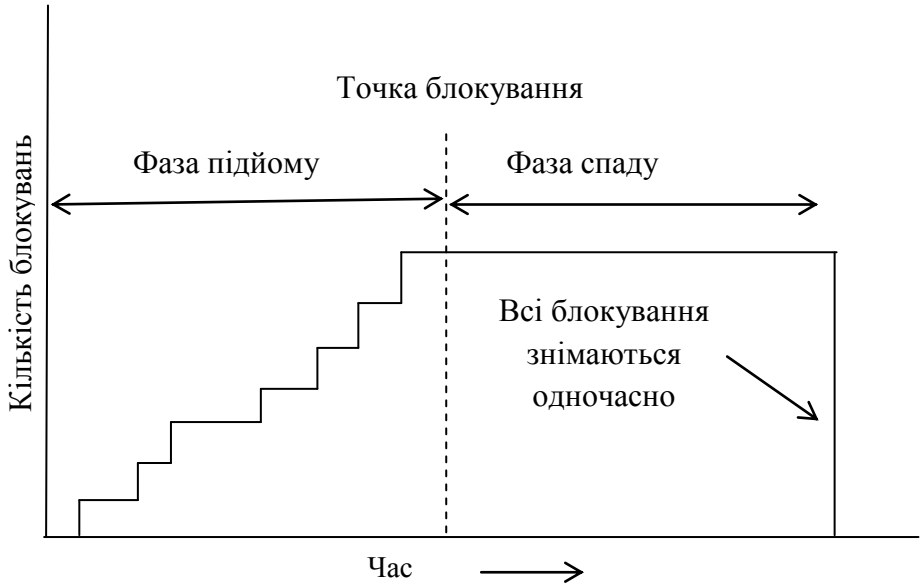


Рис. 6.22. Строге двофазне блокування

Як двофазне блокування, так і строге двофазне блокування можуть зумовити тупики. Якщо два процеси намагаються заблокувати одну пару блоків, але в різному порядку, то результатом буде **взаємне блокування**, або **тупик** (*deadlock*). У цьому разі використовуються традиційні технології запобігання тупиків, зокрема виконання блокувань у деякому строго заданому порядку. Також можливе виявлення взаємних блокувань за рахунок побудови **повного графу транзакцій**, що показує, який процес яке блокування створив і яке хоче створити, з подальшою перевіркою цього графу на цикли. Коли заздалегідь відомо, що ніяке блокування не може за нормальної роботи тривати більше t секунд, можна використати схему з **тайм-аутом**: якщо блокування неперевно належить процесу більше t секунд, то, ймовірно, відбулося взаємне блокування.

Розрізняють кілька способів реалізації двофазного блокування в розподілених системах:

1. Дані розподілені по декількох машинах. У разі **централізованого двофазного блокування** (*centralized two-phase locking*) за установку і зняття

блокувань відповідає одна машина. Усі менеджери транзакцій взаємодіють з одним централізованим менеджером блокувань, який приймає від них запити на блокування. Після того як блокування встановлено, менеджер транзакцій працює безпосередньо з менеджерами даних, далі менеджер транзакцій повертає блокування менеджерові блокувань.

2. У разі **первинного двофазного блокування** (*primary two-phase locking*) з кожного елемента даних робиться первинна копія. Блокування встановлює і знімає менеджер блокувань тієї машини, на якій розміщена копія. Первинне двофазне блокування працює фактично так само, як і централізоване, за винятком того, що може бути розподілено по декількох машинах.

3. У разі **розподіленого двофазного блокування** (*distributed two-phase locking*) передбачається, що дані можуть бути розподілені по декількох машинах. На відміну від первинного й централізованого двофазного блокування, у разі розподіленого двофазного блокування планувальники кожної з машин відповідають не тільки за установку і зняття блокувань, але й за пересилання операцій менеджерам даних (локальним), тому розподілене двофазне блокування значно ближче до базової схеми двофазного блокування, але виконується на всіх машинах, які містять дані.

Песимістичне впорядкування за відмітками часу. Абсолютно інший підхід до керування паралельним виконанням полягає в тому, щоб у момент початку кожної з транзакцій *T* призначати їй відмітку часу *ts(T)*. Для гарантування унікальності відміток часу використовуємо алгоритм Лампорта. Кожна операція, що є частиною транзакції *T*, також одержує відмітку часу *ts(T)*. Кожний елемент даних *x* у системі отримує відмітку часу читання $ts_{RD}(x)$ та відмітку часу запису $ts_{wr}(x)$. Відмітка часу читання відповідає відмітці часу транзакції, що останньою зчитувала *x*, а відмітка часу запису відповідає відмітці часу транзакції, яка останньою записувала *x*. У разі використання впорядкування за відмітками часу, якщо дві операції вступають у конфлікт, то менеджер даних виконує спочатку операцію з найменшою відміткою.

Якщо планувальник отримує від транзакції *T* з відміткою часу *ts* операцію *read (T,x)*, причому $ts < ts_{WR}(x)$, тобто планувальник повідомляється, що операція запису *x* була виконана до початку виконання *T,* то у цьому разі

транзакція *T* просто переривається. Натомість якщо $ts > ts_{WR}(x)$, тобто все правильно і немає конфлікту, то операція читання виконується. Крім того, $ts_{RD}(x)$ установлюється у значення $max\{ ts, ts_{RD}(x) \}$.

Так само виконується процедура отримання планувальником від транзакції *T* з відміткою часу *ts* операції запису *write (T,x)*. Якщо $ts < ts_{RD}(x)$, то варто лише перервати транзакцію, оскільки поточне значення *x* було зчитано попередньою транзакцією, а транзакція *T* запізнилася. Натомість якщо $ts > ts_{RD}(x)$, то, оскільки ніяка попередня транзакція не зчитувала значення *x*, можливо змінити його. Аналогічно $ts_{WR}(x)$ установлюється в $max \{ts, ts_{WR}(x) \}$.

Властивості алгоритмів відміток часу та блокувань різні. Коли транзакція працює з великими (застарілими) відмітками часу, вона переривається, блокування ж в аналогічних умовах або очікує, або може бути виконаним одразу. Під час роботи з відмітками часу не відбуваються взаємні блокування.

Впорядкування за допомогою відміток часу є у декількох варіантах, серед яких слід відзначити **стале** та **багатоваріантне** впорядкування.

Оптимістичне впорядкування за відмітками часу. Третій підхід до виконання декількох транзакцій одночасно – це **оптимістичне впорядкування за відмітками часу**. Ідея, яку покладено в основу цього підходу, проста: ідіть уперед і робіть те, що бажаєте, не звертаючи уваги, що ще щось відбувається одночасно. Проблеми вирішувати будемо тоді, коли вони виникнуть, бо на практиці конфлікти відбуваються не часто.

Конфлікти можуть бути нечастими, але вони все-таки трапляються, і необхідний спосіб їх розв’язання. Оптимістичне керування відслідковує випадки читання й запису елементів даних. У момент підтвердження транзакції перевіряється, чи не був з моменту її початку змінений якийсь елемент даних, потрібний іншим транзакціям: якщо був, то транзакція переривається, якщо не був, то підтверджується.

Оптимістичне керування паралельним виконанням транзакцій добре реалізується на основі закритих робочих просторів. У цьому разі кожна транзакція змінює свої дані в індивідуальному порядку, не перетинаючись із іншими. У кінці роботи нові дані або приймаються, або відкидаються, що зумовлює порівняно просту і зрозумілу схему роботи.

Суттєва перевага оптимістичного керування паралельним виконанням транзакцій полягає в тому, що досягається максимальний паралелізм і не буває взаємних блокувань, оскільки жодний процес ніколи не чекає завершення блокування. Недолік такого керування полягає у тому, що часом воно дає збій, і доводиться починати транзакцію знову. В умовах високого завантаження ймовірність збоїв може істотно підвищитися, що зробить оптимістичне керування не придатним для використання.

5.6.6. Компоненти розподілених транзакцій

Для фіксації розподіленої транзакції всі ланцюжки подій, які входять до складу транзакції, мають бути успішно завершені. Структурна схема (рис. 6.23) ілюструє зв'язки між активними складовими частинами варіанта середовища для виконання розподілених транзакцій.

Приклад. Розглянемо компоненти розподілених транзакцій на прикладі СУБД SQLBase 8.0.

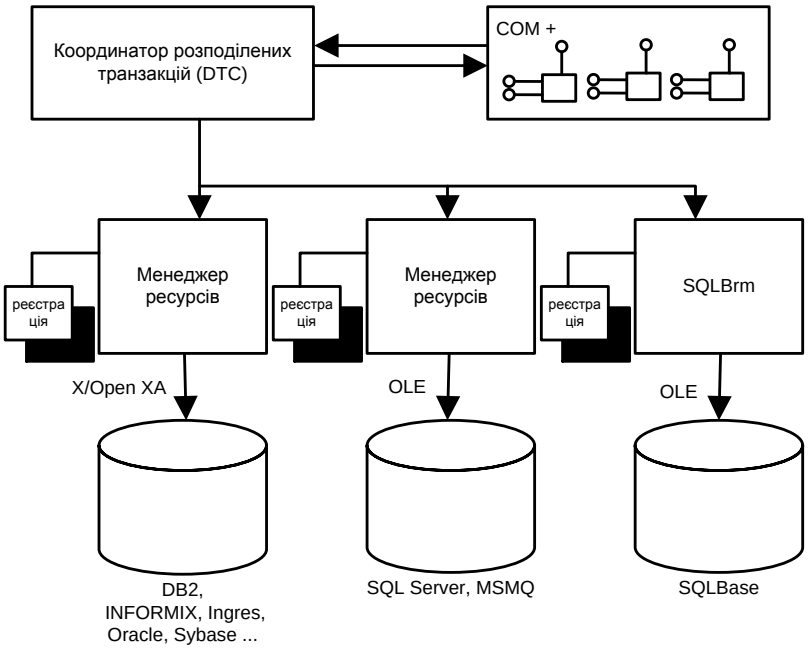


Рис. 6.23. Структурна схема зв'язків між активними складовими частинами середовища для виконання розподілених транзакцій

Координатор розподілених транзакцій (DTC). Компанія Microsoft внесла до складу ОС Windows, починаючи з Windows 2000, інструмент Distributed Transaction Coordinator (DTC). Призначенням цього засобу адміністрування є координація роботи

компонентів, які беруть участь у роботі транзакції. Для кожного комп'ютера, залученого до роботи транзакції, DTC виконує роль диспетчера транзакцій. Основним завданням такого диспетчера є реєстрація подій і компонентів, які беруть участь у роботі транзакції, та забезпечення транзакції унікальним ідентифікатором, який може застосовуватися для позначення компонентів транзакції. Крім того, диспетчер транзакцій відповідає за керування результатами транзакцій. Після успішного завершення кожної з подій і транзакції в цілому диспетчер виконує процедуру двофазної фіксації транзакції, яка виконується у такі дві фази:

Фаза 1. Диспетчер транзакцій запитує статус кожного із залучених у транзакцію компонентів (очікуваний успіх/невдача операції), залежно від статусу яких друга фаза фіксації може розвиватися за одним з двох можливих сценаріїв.

Фаза 2 (сценарій 1). Якщо кожен з опитаних компонентів повідомляє про успіх операції, то диспетчер фіксує новий статус для кожного компонента і для всієї транзакції в цілому.

Фаза 2 (сценарій 2). Якщо один або кілька компонентів, які беруть участь у транзакції, повідомляють про помилку під час виконання операції, диспетчер транзакцій скасовує всю транзакцію, що може спричинити знищення змін, які вносяться до вмісту баз даних.

SQLBrm – адміністратор ресурсів SQLBase. Адміністратор ресурсів SQLBrm відповідає за керування таким ресурсом, як база даних Gupta SQLBase; забезпечує операції з відновлюваним ресурсом, який має можливість працювати подібно до транзакції та відповідає всім властивостям ACID; адмініструє комунікаційні потоки, створюючи їх або переводячи в режим «сну» / «активний» (стосовно раніше створених потоків, які не активні зараз). Цей потік може використовуватися для підключення до сеансу серверного або клієнтського комунікаційного потоку.

SQLBrm також направляє СУБД команду фіксації/скасування транзакції, а DTC дізнається про статуси компонентів, і якщо надходить повідомлення про успіх/невдачу, то DTC координує свої дії із SQLBrm, який фактично виконує процедури фіксації/скасування стосовно бази даних SQLBase.

Крім того, SQLBrm керує комунікаційними потоками між клієнтом і сервером, використовуючи їх у разі потреби і відключаючи в періоди, коли така необхідність відпадає.

У разі відключення SQLBase SQLBrm керує процесом блокування комунікаційних потоків, відключаючи приймач інформації («Listener»), що припиняє наявні активні потоки і завершує всі сеанси.

Порядок виконання розподіленої транзакції. Зазвичай зв'язок із SQLBase через SQLBrm утворюють SQLBase OLE DB Provider або драйвер ODBC. Контакт з OLE DB Provider або драйвером ODBC (Open Database Connectivity) зумовлює створення сеансу зв'язку програмою Listener і відправлення адміністратору ресурсів запиту на підключення

потоків від клієнта й сервера до цього сеансу. Якщо наявні незайняті потоки, то адміністратор використовує їх, інакше для клієнта і сервера створюються нові комунікаційні потоки.

Клієнту надсилається повідомлення про прийняття запиту, після чого він надсилає запит на з'єднання. Цей запит направляється через потік клієнта в потік сервера і потім надходить у SQLBase. Відповідь на запит проходить той самий шлях у зворотному напрямку.

Якщо клієнт використовує транзакцію MTS (Microsoft Transaction Server), SQLBrm отримає інструкцію на включення цієї транзакції, після чого SQLBrm установлює статус сеансу «активний». Будь-яка операція з базою даних виконується в такій самій послідовності, що і запит на підключення.

Коли клієнт фіксує транзакцію *MTS, DTC* спочатку повідомляє всіх задіяних адміністраторів ресурсів про необхідність підготовки до фіксації, а потім здійснює фіксацію, SQLBrm установлює відмітку фіксації сеансу і видаляє дані з журналу. У разі розриву з'єднання з клієнтом у SQLBase відправляється відповідне повідомлення, і клієнтський потік відключається від сеансу. Після прийняття SQLBase повідомлення про припинення з'єднання виконується відключення серверного потоку і видалення сеансу.

Час виконання транзакції *MTS* може перевищувати тривалість кількох підключень або виконання інших транзакцій *MTS*, а також може бути етапом обробки одного підключення. У цьому полягає відмінність від звичайного для SQLBase визначення транзакції як події, яка відбувається в період від підключення до відключення.

5.7. Висновки

1. Взаємодія між процесами пов'язана із синхронізацією процесів у розподілених системах. Проблема розподілених систем і комп'ютерних мереж загалом полягає в тому, що для них немає поняття єдиних годин, які спільно використовуються, тобто процеси на різних машинах мають власний час.

2. Наявні різні способи синхронізації годин у розподілених системах, але всі вони грунтуються на обміні показниками годинників, що вимагає враховувати затримки на надсилання й одержання повідомлень. Точність алгоритмів синхронізації значною мірою залежить від варіації в затримках доступу і способів обліку цих варіацій.

3. Зазвичай знання абсолютного часу не потрібно, а достатньо, щоб відповідні події в різних процесах відбувалися у правильній послідовності.

Лампорт показав, що, увівши поняття «логічні години», можна змусити набір процесів дотримуватися загальної угоди щодо правильної черговості подій. Кожна подія, зокрема надсилання або прийом повідомлення, одержує абсолютно унікальну логічну відмітку часу *C(a)*. Якщо подія *a* відбувається раніше *b*, то $C(a) < C(b)$. Поняття «відмітка часу Лампорта» було вдосконалено до векторних відміток часу: якщо $C(a) < C(b)$, то подія *a* передує події *b*.

4. Оскільки в розподілених системах немає поняття «спільно використовувана пам'ять», часто важко визначити поточний стан системи. Глобальний стан розподіленої системи можна визначити за допомогою синхронізації всіх процесів так, щоб кожний з них визначив і зберіг свій локальний стан разом з повідомленнями, які передаються в цей момент. Синхронізація може бути виконана без зупинки процесів і запису їх стану. Замість цього під час роботи розподіленої системи з неї можна зробити миттєвий зліпок – розподілений знімок стану.

5. Синхронізація між процесами часто вимагає, щоб один з процесів виконував роль координатора. У цьому разі якщо координатора не визначено попередньо, то необхідно, щоб процеси в розподілених обчисленнях могли вирішити, хто з них буде координатором. Це рішення приймається за допомогою алгоритмів голосування, які переважно задіюються у разі збоїв або відключень наявного координатора.

6. Важливий клас алгоритмів синхронізації – розподілені взаємні виключення. Ці алгоритми гарантують, що в розподіленому наборі процесів доступ до спільно використовуваних ресурсів має хоча б один процес. Розподілені взаємні виключення можна легко забезпечити за допомогою координатора, який відстежуватиме, чия зараз черга. Є також і повністю розподілені алгоритми, але їх недолік – надто велика чутливість до збоїв процесів і зв'язку.

7. З процесами взаємного виключення пов'язані розподілені транзакції, які є набором операцій з даними, що сумісно використовуються, при цьому вся транзакція або повністю виконується, або не виконується зовсім. Декілька транзакцій можуть виконуватися одночасно, а результат є таким,

неначе ці транзакції виконувалися в деякій заданій послідовності. Транзакції довговічні, тому що після закінчення їх результат незмінний.

5.8. Запитання для самоконтролю

1. Як процеси синхронізуються між собою?
2. Де, навіщо потрібна синхронізація?
3. Що таке синхронізація часу?
4. Що таке фізичні годинники?
5. Які дві основні проблеми характерні для систем реального часу?
6. Як розраховують середній сонячний день?
7. Що таке TAI?
8. Що таке UTC?
9. Що таке *mean solar second*?
10. Що таке атомний годинник?
11. Що таке *leap seconds*?
12. Пояснити алгоритми синхронізації годин.
13. Що таке логічні годинники?
14. Який алгоритм запропонував Лампорт?
15. Дати визначення відмітки часу Лампорта.
16. Дати визначення векторної відмітки часу.
17. Що таке глобальний стан?
18. Що таке алгоритми голосування? Які вони бувають?
19. Пояснити алгоритм забіяки.
20. Пояснити кільцевий алгоритм.
21. Пояснити поняття «взаємне виключення».
22. Пояснити централізований алгоритм.
23. Пояснити розподілений алгоритм.
24. Пояснити алгоритм маркерного кільця.
25. Порівняти три алгоритми (централізований, розподілений та алгоритм маркерного кільця).
26. Навести приклад однієї з моделей транзакцій.

27. Які примітиви використовуються у транзакціях?

28. Дати класифікацію транзакцій.

29. Навести два методи реалізації транзакцій.

30. Що таке закритий робочий простір, де його використовують?

31. Що таке журнал із записом, де його використовують?

32. Яка мета керування паралельним виконанням транзакцій?

33. Пояснити модель контрольних точок та принцип її роботи.

34. Пояснити модель багатоланкових транзакцій.

35. Порівняти моделі контрольних точок та багатоланкових транзакцій.

36. Що таке рівень ізольованості транзакцій? Які рівні ізольованості транзакцій використовують на практиці?

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

IP	Internet Protocol – міжмережний протокол
ТКС	Телекомунікаційні системи
ТСМ	Телекомунікаційні системи та мережі
CRM – система	Customer relationship management – система управління взаємовідносинами з клієнтами
TCP/IP	Transmission Control Protocol/Internet Protocol – протоколи керування передачею даних у мережах
HTTP	HyperText Transfer Protocol – протокол передачі гіпертексту
OSI	Open Systems Interconnection – модель взаємодії відкритих систем
DoD	Модель взаємодії комп'ютерних систем, модель DoD
EDP	Electronic Data Processing - інформаційно-телекомунікаційні системи електронної обробки даних
EDI	Electronic Data Interchange - обмін даних на основі Intranet-технологій
RPC	Remote Procedure Call – протокол виклику віддалених процедур
CPU	Central processing unit – центральний процесор
СУБД	Система управління базами даних
SISD	Single Instruction Single Data – архітектура, яка використовує один потік команд, один потік даних
MIMD	Multiple Instruction Multiple Data – архітектура, яка використовує багато потоків команд, багато потоків даних
MISD	Multiple Instruction Single Data – архітектура, яка використовує множинні інструкції, єдиний потік даних
SIMD	Single Instruction Multiple Data – архітектура, яка використовує один потік команд, багато потоків даних

SMP	Symmetrical Multiprocessing – симетричні мультипроцесори
MPP	Massively Parallel Processing – масово паралельні системи
FDDI	Fiber Distributed Data Interface – інтерфейс для передачі розподілених даних волоконно-оптичними каналами
PCI SCSI	Peripheral Component Interconnect Small Computer System Interface –архітектура підключення периферійних компонентів малих комп'ютерних систем (сказі-інтерфейс)
DEC	Digital Equipment Corporation – назва корпорації
NCR	National Cash Register – назва корпорації
AWS	Технологія, розроблена American Welding Society для виготовлення дисків
API	Application programming interface – інтерфейс прикладного програмування (набір функцій, які надаються для використання у прикладних програмах)
AMD	Advanced Micro Devices Corporation – назва корпорації
HOST	Центральний комп'ютер
IBM PC	Тип персонального комп'ютера корпорації IBM
DNS	Domain Name System – служба доменних імен
БД	База даних
ПК	Персональний комп'ютер
ATM	Asynchronous Transfer Mode – асинхронний режим передачі даних, протокол
INMOS	Назва корпорації, яка виготовляє трансп'ютери
CC	Cognitive Compute – розумний комп'ютер, умовне позначення
AIX	Advanced Interactive Executive – операційна система
BP	Base Pointer – вказівник бази, регістр у процесорах 80x86
ОС	Операційна система
DOS	Distributed Operating System – розподілена операційна система

NOS	Network Operating Systems – мережна операційна система
RAS	Remote Access Service – сервіс віддаленого доступу
DSM	Distributed Shared Memory – розподілена, поділювана (спільно використовувана) оперативна пам’ять
UDP	User Datagram Protocol – протокол дейтаграм користувача
ІК	Інтерфейс користувача
ЛП	Логіка прикладних програм
ДД	Доступ до даних
POSIX	Portable Operating System Interface for Computer Environment for Unix – інтерфейс операційної системи, яка переноситься
CLI	Call Level Interface – прикладний програмний інтерфейс рівня викликів
OSI/ISO	Open Systems Interface/ International Organization for Standardization – інтерфейс відкритих систем, еталонна модель взаємодії відкритих систем, розроблена та затверджена Міжнародною організацією зі стандартизації
RMI	Remote method invocation – протокол, технологія побудови розподіленого прикладного програмного забезпечення у специфікації мови Java
XML	Extensible Markup Language – мова розмітки текстового документа
DTD	Document Type Definition – визначення типу документа, опис типу (шаблона) документа
XSD	Extensible Schema Definition – визначення розширюваної схеми XML
SOAP	Simple Object Access Protocol – простий протокол доступу до об’єктів
WSDL	Web Service Definition Language – мова опису Web – сервісів
CLR	Common Language Runtime - загальна система підтримки часу виконання, загальномовне середовище виконання, віртуальна машина CLR

CGI	Common Gateway Interface – загальний шлюзовий інтерфейс
PHP	Personal Home Page / Hypertext Preprocessor – мова розробки персональних базових сторінок / гіпертекстовий препроцесор
ASP	Active Server Pages / Application Service Providers – мова розробки серверних прикладних програм / концепція оренди прикладного програмного забезпечення з розміщенням усієї серверної інфраструктури у провайдера цих послуг
JSP	Java Server Pages – серверні сторінки Java
JVM	Java Virtual Machine – віртуальна Java-машина
ISAPI	Internet Server API – інтерфейс прикладного програмування Internet -сервера
IIS	Internet Information Server – інформаційний сервер Internet компонента Windows NT Server
ADO	ActiveX Data Objects – об’єкти даних ActiveX, механізм (технологія) ADO, набор компонент ActiveX, які використовуються для доступу до бази даних, що підтримує специфікацію OLE DB
MSIL	Microsoft Intermediate Language – у технології Microsoft .NET є мовою проміжного рівня
RIA	Rich Internet Applications – технологія створення інтерфейсу користувача з активними компонентами
UI	User Interfase – інтерфейс користувача
КПК	Кишенькові персональні комп’ютери
MXML	Macromedia eXtensible Markup Language – декларативна мова опису інтерфейсів, яка ґрунтується на XML, використовує платформу Adobe Flex для розробки RIA насичених Internet – прикладних програм
CSS	Cascading Style Sheets – каскадні таблиці стилів
IDE	Integrated Development Environment – інтегроване середовище розробки

Adobe AIR	Adobe Integrated Runtime – середовище виконання Web – прикладних програм
PDF	Portable Document Format – формат документів
NPAPI	Netscape Plugin Application Programming Interface – крос-платформна архітектура розробки плагінів, яка підтримується багатьма браузерами
SWF	Shockwave Flash – Adobe Flash формат файлів, які використовують для multimedia
WPF	Presentation Foundation – технологія Windows для створення високо інтерактивного та насиченого інтерфейсу користувача
DOM	Document Object Model – об'єктна модель документів
SQLite	Система керування базами даних Google Gears
IE	Internet Explorer – браузер
WMV	Windows Media Video – кодак для перегляду мультимедійних файлів
XAML	eXtensible Application Markup Language – декларативна XML-базована мова
GDI	Graphics Device Interface – компонент ядра операційної системи Microsoft Windows для роботи з графікою
WMA	Windows Media Audio – цифровий формат аудіо файлів, розроблений Microsoft
MP3	Формат перетворення аудіоданих
DLR	Dynamic Language Runtime – сервіс підтримки динамічної мови CLR (Common Language Runtime)
FX	Software Framework – абстрактне середовище розробки, виконання прикладного програмного забезпечення
SDK	Software Development Kit – інструментарій розробки прикладного програмного забезпечення
IDE	Integrated development environment – програмне середовище, яке забезпечує повні можливості розробки програмного

	забезпечення
SE	Standard Edition – стандартна редакція, позначення, яке трапляється в назвах версій програмних продуктів
ME	Micro Edition – підмножина програмної платформи для пристроїв, які мають обмежені ресурси(сотові телефони, карманні персональні комп'ютери та інші)
AJAX	Asynchronous Java Script and XML – асинхронний Java Script та XML, підхід до побудови інтерактивних інтерфейсів користувача Web-орієнтованого прикладного програмного забезпечення, який полягає у «фоновому» обміні даними браузера з Web-сервером
J2EE	Java Platform Enterprise Edition - набір специфікацій та відповідної документації мови Java, який описує архітектуру серверної платформи для задач середніх та великих підприємств
EJB	Enterprise JavaBeans – специфікація технології створення та підтримки серверних компонентів, які містять бізнес-логіку
Web UI	Web User Interface – інтерфейс користувача у Web-середовищі
JTA	Java Transaction Application programming interface- Java API для транзакцій, визначає взаємодію між менеджером транзакцій та іншими учасниками розподіленої системи транзакцій
HTTPS	(Hypertext Transfer Protocol Secure) – розширення протоколу HTTP, яке підтримує шифрування
URI	Uniform Resource Identifier – уніфікований (однаковий) ідентифікатор ресурсу
EIS	Executive Information System – керівна інформаційна система
IIOP	Internet Inter-ORB (Object Request Broker) – Protocol прото-

	кол, який визначає передачу повідомлень між об'єктами по TCP/IP
JDBC	Java Database Connectivity – Java-інтерфейс для роботи з базами даних
VB	Мова програмування Visual Basic
XSD	XML Schema – мова опису структури XML-документа
WSDL	Web Services Description Language – мова опису Web-сервісів та доступу до них
ISO	International Organization for Standardization – міжнародна організація стандартизації
ADSL	Asymmetric Digital Subscriber Line – асиметрична цифрова абонентська лінія, модемна технологія, в котрій доступна полоса пропускання каналу, розподілена між вхідним та вихідним трафіками асиметрично
ISDN	Integrated Services Digital Network – цифрова мережа з інтеграцією обслуговування, яка дозволяє поєднати послуги телефонного зв'язку та обміну даними
SONET	Synchronous Optical Networking – технологія транспортних телекомунікаційних мереж
ITU V	International Telecommunications Union – Международний союз електрозв'язку, стандартні модемні протоколи ITU
ARCNET	Attached Resources Computer network – мережа ARCnet є мережною архітектурою локальної мережі
CDP	Cisco Discovery Protocol – протокол другого рівня, розроблений компанією Cisco Systems, який дозволяє здійснювати підключення (напряму або через пристрої першого рівня) мережного обладнання Cisco, його назву, версію та IP-адресу
DCAP	Data Link Switching Client Access Protocol – протокол клієнтського доступу до комутації каналів передачі даних
DTP	Distributed Transaction Processing – розподілена обробка транзакцій

L2FP	Layer 2 Forwarding Protocol (L2F) – протокол естафетної передачі на другому рівні, один з протоколів тунелювання, розроблений компанією Cisco Systems для створення віртуальних приватних мереж зв'язку через Internet
L2TP	Layer 2 Tunneling Protocol - протокол L2TP мережний протокол тунелювання канального рівня
CLNP	Connection Less Network Protocol - безконтактний мережний протокол, дейтаграмний протокол передачі даних, схожий на IP, який використовують в Internet
IPv4	Internet Protocol версії 4 – протокол, який використовують в Internet
IPv6	Internet Protocol версії 6 – протокол, який використовують в Internet
IGRP	Interior Gateway Routing Protocol – протокол маршрутизації, розроблений фірмою CISCO для своїх багато протокольних маршрутизаторів
Sec	Internet Protocol Security – набір протоколів для забезпечення захисту даних, які передаються міжмережним протоколом IP, дозволяє підтверджувати дійсність і/або шифрування IP-пакетів
TP0	Transport Protocol Class 0 (Simple Class) – найпростіший транспортний протокол OSI
TP4	Transport Protocol Class 4 (Error Detection and Recovery Class) – транспортний протокол OSI класу 4 з перевіркою правильності передачі та виправленням помилок
FTP	File Transfer Protocol – протокол передачі файлів у комп'ютерних мережах
XMPP	Extensible Messaging and Presence Protocol – протокол обміну повідомленнями та інформацією про наявність, який розширюється
UMSP	Unifired Memory Space Protocol – уніфікований протокол

	розподілу простору пам’яті
RPC	Remote Procedure Call – клас технологій, які дозволяють комп’ютерним програмам викликати функції або процедури в іншому адресному просторі
QoS	Quality of Service – якість обслуговування трафіку в каналі зв’язку
RSVP	Resource Reservation Protocol – протокол резервування мережних ресурсів
FIFO	First in, First out – принцип обслуговування черг «першим прийшов – першим обслужений»
RED	Random Early Detection – довільне раннє обслуговування, один з алгоритмів керування переповненням черг маршрутизаторів
CBQ	Class Based Queuing – черги, які приєднані до інтерфейсу та створюють дерево
WFQ	Weighted Fair Queuing – чесна черга з ваговими коефіцієнтами, механізм планування пакетних потоків даних із різними пріоритетами
RESV	Тип повідомлення протоколу RSVP, яке генерується одержувачами й поширюється вгору деревом маршрутизації, передається кожному маршрутизатору на шляху потоку
PATH	Тип повідомлення протоколу RSVP, яке використовується для поширення інформації про зворотний маршрут
IETF	Internet Engineering Task Force – відкрите міжнародне співтовариство, яке займається розвитком протоколів та архітектури Internet
RFC	Request for Comments – документ із серії пронумерованих інформаційних документів Internet, які вміщують технічні специфікації та стандарти
JID	Jabber ID – ідентифікатор протоколу Jabber (нині XMPP)

SMS	Short Message Service – служба коротких повідомлень, технологія, яка дозволяє приймати й передавати короткі текстові повідомлення
VM	Virtual Machine – програмне або апаратне середовище, яке виконує деякий код (байт-код, машинний код реального процесора), або специфікації деякої системи (віртуальна машина мови програмування)
MMU	Memory Management Unit – блок керування пам'яттю, компонент апаратного забезпечення комп'ютера, який відповідає за доступ до пам'яті
TLB	Translation lookaside buffer – технологія, яка оптимізує перетворення віртуальних адрес у фізичні
UNIX	Сімейство багатозадачних багатокористувацьких операційних систем, які є переносними
IPC	Inter Proccess Communication – міжпроцесна взаємодія, набір способів обміну даними між множиною потоків у одному або більше процесах
IANA	Internet Assigned Numbers Authority – «Адміністрація адресного простору», американська некомерційна організація, яка керує просторами IP-адрес, доменами верхнього рівня, а також реєструє типи даних стандарту багатоцільового розширення Internet-пошти й параметри інших протоколів
DCE	Distributed Computing Environment – середовище розподілених обчислень
URL	Uniform Resource Locator – універсальний локатор ресурсів, стандартизований спосіб запису адреси ресурсу в мережі Internet
TCL	Tool Command Language – «командна мова інструментів», скрипкова мова високого рівня
FIPA	Foundation for Intelligent Physical Agents – організація, яка

	розробила узагальнену модель програмних агентів
ACC	Agent Communication Channel – канал зв’язку між агентами
IIOP	Internet INTER-ORB Protocol – абстрактний протокол для TCP/IP у розподілених об’єктних системах, який забезпечує інтероперабільність брокерів.
ACL	Agent Communication Language – мова взаємодії програмних агентів
CFP	Computer-aided financial planning – комп’ютеризоване фінансове планування
DNS	Domain Name System – система доменних імен, комп’ютерна розподілена система для отримання інформації про домени
FQDN	Fully Qualified Domain Name – «повністю визначене ім’я домену», ім’я домену, яке не має неоднозначностей у визначенні
BIND	Berkeley Internet Name Domain – відкрита й найбільш поширена реалізація DNS – сервера, яка забезпечує виконання перетворення DNS – імені в IP- адресу і навпаки
SOA	Service-oriented architecture – модульний підхід до розробки програмного забезпечення, який використовує розподілені слабо зв’язані компоненти, що замінюються та мають стандартизовані інтерфейси для взаємодії за стандартними протоколами
MX	Mail Exchanger – один з типів записів у DNS, який вказує спосіб маршрутизації електронної пошти
SRV	SRV – запис, серверний запис, стандарт в DNS, який визначає місцезнаходження (ім’я хоста і номер порту) серверів певних служб
NS	Name server – назва запису в DNS, яка вказує на DNS – сервер (сервер імен) для певного домену або скорочена

	назва DNS – сервера
CNAME	Canonical Name Record – канонічний запис імені (псевдонім), який використовується для перенапрям на інше ім’я
PTR	Pointer – назва рекурсного запису DNS, що зв’язує ім’я хоста з канонічним іменем хоста (використовується для реалізації зворотного DNS)
HINFO	Host Information – інформація про вузол, хост
TXT	Розширення файлу, яке визначає файл, що містить текстові дані, організовані у вигляді рядків
DIB	Device Independent Bitmap – формат файлу, за якого програміст може отримати доступ до всіх елементів структур, які описують зображення, за допомогою звичайного вказівника
RDN	Relative Distinguished Name – унікальне ім’я, яке складається з одного або декількох відносних унікальних імен, розділених комою
DIT	Directory Information Tree – інформаційне дерево каталогів, за допомогою якого утворюють граф іменування служби каталогів X.500, у якому кожний вузол є елементом каталогу та може мати кілька дочірніх вузлів, для яких він буде батьком
DSA	Directory Service Agent – агент служби каталогів, програмний засіб, що забезпечує роботу з інформаційним фондом
DUA	Directory User Agent – агент каталогу користувача, програмний процес
LDAP	Lightweight Directory Access Protocol – «полегшений протокол доступу до каталогів», протокол прикладного рівня для доступу до служби каталогів X.500, розроблений IETF
ARP	Address Resolution Protocol – протокол визначення адреси, протокол мережного рівня, призначений для встановлення MAC-адреси за відомою IP-адресою

COW	Тип паралельного комп'ютера, який використовує принцип Copy-On-Write, оптимізує багато процесів операційної системи, таких як робота з пам’яттю або файлами на диску
MPP	Тип паралельного комп’ютера, який використовує масово-паралельну архітектуру (Massive Parallel Processing) обчислювальної системи
RMI	Remote Method Invocation – програмний інтерфейс виклику віддалених методів мови Java
SSP chains	Stub-Scion Protocol chains - ланцюжок вказівників, які пересилаються, реалізується у вигляді пари (замісник, скелетон)
TAI	Internatoinal Atomic Time – міжнародний атомний час
UTC	Universal Coordinated Time – універсальний скоординований час, середній час за Гринвічем
NIST	National Institute of Standards and Technology – Національний інститут стандартів і технологій (США)
WWV	Радіостанція з високочастотними позивними Форту Коллінз, Колорадо, США
MSF	Англійська радіостанція, яка використовує MSF-сигнал для передачі високоточного часу
GEOS	Geostationary Environment Operational Satellite – геостаціонарний мульти – експериментальний супутник для магнітосферних досліджень
NTP	Network Time Protocol – протокол та програмне забезпечення, впроваджене для синхронізації годинників комп’ютерних систем з пакетною комутацією та змінною затримкою передачі даних мережею

СПИСОК ЛІТЕРАТУРИ

1. **Tanenbaum A.** Computer Networks. Englewood Cliffs/ Tanenbaum A. - NJ: Prentice Hall, 3rded., 1996.

2. **Э.Таненбаум** Распределенные системы: принципы и прадигмы/ Э.Таненбаум – Питер, 2003. – 977 стр.

3. **Дейтел Х.М.** Операционные системы. Распределенные системы, сети, безопасность: учебное пособие/ Дейтел Х.М. - 3-е изд. – Бином-пресс, 2011. – 704 с.

4. **С. Морган** Разработка распределенных приложений на платформе Microsoft .Net Framework/ С. Морган, Б. Райан, Ш. Хорн, М. Бломсма, 2008 – 608 с.

5. **Wang H.,** Consumer Privacy Concerns about Internet Marketing/ Wang H., LO, M. K., Wang C. – Commun : ACM, 1998.

6. **Ноутон П.,** Java 2/ Ноутон П., Шилдт Г. – Петербург: БХВ, 2001.- 1102 с. -ISBN: 978-5-94157-012-6, 0-07-211976-4.

7. **Blair G.** Open Distributed Processing and Multimedia/ Blair G., Stefani J.-B.- Addison Wesley 1 edition, 1997.- p.480 - ISBN-10: 0201177943, ISBN-13: 978-0201177947.

8. **B.Cliord Neuman** Scale in Distributed Systems// B.Cliord Neuman - In Casavant T., Singhal M. (eds.). Readings in Distributed Computing Systems, Los Alamitos, C A: IEEE Computer Society Press, 1994.- p.29

9. **Lilja D.** Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issuesand Comparisons// Lilja D. - ACM Computing Surveys, Vol. 25, No. 3, September 1993. - pp. 303-338.

10. **Bhoedjang R.** User-Level Network Interface Protocols// Bhoedjang R., Ruhl T., Bal H. IEEE Computer, 1998. – pp. 53 – 60.

11. **Day J.** The OSI Reference Model.// Day J., Zimmerman H. Proceedings of the IEEE, Dec. 1983.

12. **Comer D**. Internetworking with TCP/IP// Comer D., Volume I: Principles, Protocols, and Architecture. Upper Saddle River, NJ: Prentice Hall, 4th ed., 2000.

13. **Handel R.** ATM Networks// Handel R., Huber M., Schroder S. Wokingham, Addison- Wesley, 2nd ed., 1994.

14. **Schulzrinne, H.** RTP: A Transport Protocol for Real-Time AppUcations/ Schulzrinne, H., Casner S., Frederick R., Jacobson V. - RFC 1889, Jan. 1996 – p.104

15. **Lewis B.** Multithreaded Programming with Pthreads. Englewood Cliffs/ Lewis B., Berg D.J. NJ: Prentice Hall, 2nd ed., 1998.

16. **Stevens W.** UNIX Network Programming – Interprocess Communication/ Stevens W. – Prentice Hall Ptr, 1998 – 556 с.

17. **Triantafillou P.** Achieving Strong Consistency in a Distributed File System// Triantafillou P., Neilson C. - IEEE Trans. Softw. Eng., Jan. 1997 – pp. 35-55.

18. **Katz E.** A Scalable HTTP Server: The NCSA Prototype// Katz E., Butler M., McGrath R. - Сотр. Netw. & ISDN Syst., Sept. 1994 – p.10.

19. **Dimitrov B.** Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms// Dimitrov B., Rego V. - IEEE Trans. Par. Distr. Syst., May 1998.

20. **Lewis B.** Multithreaded Programming with Pthreads// Lewis B., Berg D.J. – Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1998.

21. **А. Цимбал** Технологии создания распределенных систем. Для профессионалов/ А. Цимбал, М. Аншина – Питер, 2003. – 576 с.

22. **Stevens W.** UNIX Network Programming – Interprocess Communication/ Stevens W., 1999 – p. 558.

23. **Stevens W.** Advanced Prograinming in the UNIX Environment/ Stevens W., Reading, MA: Addison-Wesley, 1992 – p. 960.

24. **Stevens W.** UNIX Network Programming – Networking APIs/ Stevens W., Sockets and XTI. Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1998.

25. **Pike R.** Plan 9 from Bell Labs/ Pike R., Presotto D., Dorward S., Flandrena B., Thompson K., Trickey H., and Winterbottom P., Computing Systems, Summer 1995.

26. **Radia S.** Names, Contexts, and Closure Mechanisms in Distributed Computing Environments// Radia S., Ph.D. Thesis, University of Waterloo, Ontario, 1989.

27. **Rao H.** Accessing Files in an Internet: The Jade File System// Rao H. and Peterson L., IEEE Trans. Softw. Eng., June 1993- p.29.

28. **Lampson B.** Designing a Global Name Service// Lampson B., Proc. Fourth Symp. on Principles of Distributed Computing. ACM, 1986 – p.10.

29. **Vixie P.** A DNS RR for Specifying the Location of Services (DNS SRV)// Vixie P., RFC 2052, Oct. 1996.

30. **Katz E.** A Scalable HTTP Server: The NCSA Prototype// Katz E., Butler M., and Mcgrath R., Сотр. Netw. & ISDN Syst. 1994.

31. **Andrews G.** Foundations of Multithreaded, Parallel, and Distributed Programming/ Andrews G., Reading, MA: Addison-Wesley, 2000.

32. **Singhal M.** Advanced Concepts in Operating Systems/ Singhal M. and Shivaratri N.: Distributed, Database, and Multiprocessor Operating Systems. New York: McGraw- Hill, 1994.

33. **Wu J.** Distributed System Design. Boca Raton// Wu J., CRC Press, 1998.

34. **Drummond R.** Low-Cost Clock Synchronization// Drummond R. and Babaoglu O., Distributed Computing, 1993 – p. 10.

35. **Б. Эккель** Философия Java. Библиотека программиста/ Б. Эккель, 4-е изд. 2009 – 638 с.

36. **B. Chapman** Extending HPF for advanced data-parallel applications// B. Chapman, P. Mehrotra, H. Zima, IEEE Parallel and Distributed Technology, 1994 – pp.15-27.

37. **B.M. Maggs** Models of parallel computation: a survey and synthesis// B.M. Maggs, L.R. Matheson, and R.E. Tarjan, In Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS) volume 2, 1995 – pp. 61-70.

38. **D.Y. Cheng** A survey of parallel programming languages and tools// D.Y. Cheng, Moffett Field, 1993.

39. **Ільченко М.Ю.** Сучасні телекомунікаційні системи / Ільченко М.Ю., Кравчук С.О. – НВП "Видавництво "Наукова думка" НАН України", 2008. – 328 с.

40. **Корнейко О. В.** Основи теорії телекомунікацій : підручник / Корнейко О. В., Кувшинов О. В., Лежнюк О. П., Лівенцев С. П., Сакович Л. М.,

Уривський Л. О.; / за заг. ред. Ільченка М. Ю. - К.: Вид-во ІСЗЗІ НТУУ «КПІ», 2010 - 788 с.

41. **Бунін С.Г.** Комп'ютерні мережі з бездротовим доступом: навчальний посібник / Бунін С.Г., Олійник В.Ф., Сайко В.Г.та ін. - Київ, НІКА-Центр, 2007 – 293 с.

42. **Глоба Л.С.** Математичні основи побудови інформаційно-телекомунікаційних систем: навчальний посібник для студентів спеціальності 8.092401 «Телекомунікаційні системи та мережі» / Глоба Л.С.. - К.: НТУУ «КПИ», 2006. - 356 с.

43. **Романов А.И.** Телекоммуникационные сети и управление / Романов А.И. - Киев, ВПЦ «Киевский Университет», 2003. - 247с.

44. **Кравчук С.О.** Основи комп‘ютерної техніки: Компоненти, системи, мережі: навч. посіб. для студ. вищ. навч. закл./ Кравчук С.О., Шонін В.О., – К.: ІВЦ «Видавництво «Політехніка»: Видавництво «Каравела», 2005. – 344 с.

45. **Кравчук С.О.** Основы программирования на языке Java: Учеб. пособ. / Кравчук С.О., Шонін В.О., – К.: Норита плюс, 2007. – 280 с.

46. **Баженов В.А.** Інформатика. Компьютерна техніка. Комп‘ютерні технології: підручник / Баженов В.А., Лізунов П.П., Резніков А.С., Кравчук С.О., Шонін В.О., Дудзяний І.М., Левченко О.М., Горлач В.М., Коркуна М.Д., Венгерський П.С., Гарвона В.С., Ананьєв О.М. - 2-ге вид., К.: Каравела, 2007. – 640 с.

47. **Наритник Т.М.** Радіорелейні та тропосферні системи передачі: навчальний посібник / Наритник Т.М., Волков В.В. - ІВЦ "Політехніка", 2009р. -331с.

48. Електронное руководство FreeBSD handbook. (http://www.freebsd.org/handbook/index.html).

49. **Rob Ross** Beowulf HOWTO/ Rob Ross (http://www.linuxdoc.org/HOWTO/Beowulf-HOWTO.html).

50. **Gabriele Kotsis** Interconnection Topologies and Routing for Parallel Processing Systems/ Gabriele Kotsis http://www.ani.univie.ac.at/~gabi/papers/in.ps.gz.

51. **Ian Foster** Designing and Building Parallel Programs/ Ian Foster, Addison-Wesley, 1995.

52. **Jonathan M.D.** Hill. An introduction to the data-parallel paradigm/ Jonathan M.D., Department of Computer Science, 1994.

53. **K. Hwang** Advanced Computer Architecture: Parallelism, Scalability, Programmability/ K. Hwang, McGRAW-HILL, 1993.

54. **K. M. Chandy** Integrated support for task and data parallelism// K. M. Chandy, I. Foster, K. Kennedy, C. Koelbel, W. Tseng., Supercomputer Applications, 1994 – pp.80-98.

55. **Kai Hwang** Computer Architecture and Parallel Processing/ Kai Hwang, Faye A. Briggs, McGRAW-HILL, 1986.

56. **Mark Baker Cluster** Computing White Paper/ Mark Baker, 2000 – p.119.

57. **Marshall Kirk McKusick** The Design and Implementation of the 4.4BSD Operating System/ Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman, Addison-Wesley Longman, Inc. 1996.

58. **P. Mehrotra** Programming distributed memory architectures. In Advances in Languages and Compilers for Parallel Computing/ P. Mehrotra and J. Van Rosendale, MIT Press, 1991.

59. **Rajkumar Buyya** High Performance Cluster Computing: Architectures and Systems/ Rajkumar Buyya, Prentice Hall PTR, 1999 – p.849.

60. **V. Kumar** Scalable load balancing techniques for parallel computers// V. Kumar, A. Grama, and V. Rao, Parallel and Distributed Computing, 1994 – pp. 60-79.

61. **W. Gellerich** Massively Parallel Programming Languages - A Classification of Design Approaches// W. Gellerich, and M.M. Gutzmann, In K. Yetongnon, and S. Hariri, editors, Proceedings of the ISCA 9th International Conference on Parallel and Distributed Computing Systems vol. 1, ISCA, 1996 – pp. 110-119.

62. **Graham E. Fagg** PVMPI: An integration of the PVM and MPI systems// Graham E. Fagg, Jack J. Dongarra, Calculateurs Paralleles, 1996 – pp. 595-605.

63. **Giorgos Gousios** A comparison of portable dynamic web content technologies for the apache web server// Giorgos Gousios and Diomidis Spinellis, In Proceedings of the 3rd International System Administration and Networking Conference SANE 2002, Maastricht, The Netherlands, May 2002 – pp. 103-119.

64. **Paul Monday** Evolution or revolution, JSP pages become pivotal players in Web services/ Paul Monday, (http://www.ibm.com/developerworks/java/library/j-j%20ljsp.html?dwzone=java)

65. **Hans Bergsten** An Introduction to Java Servlets/ Hans Bergsten (http://www.webdevelopersjournal.com/articles/intro to servlets.html)

66. **Хейфец И.** Архитектура .NET/ Хейфец И. (обзор) (http://www.gotdotnet.ru/default.aspx?s=doc&d no=24&c по=4).

67. **Филев А.** Сравнивая .NET и Java// Филев А., 2010. (http://www.gotdotnet.ru/blogs/andrew-filev/6383/).

68. **Арчер Т.** Основы C#. Новейшие технологии/ Арчер Т. - М.: Издательско-торговый дом «Русская редакция», 2001 – 448 с.

69. **Старостин Д.** Новый «универсальный клей» - Web Services. Microsoft// Старостин Д., 2005. (http://esco-ecosys.narod.ru/2005_1/art13.htm).

70. **Рейли Д.** Создание приложений Microsoft ASP.NET/ Рейли Д. - М.: Издательско-торговый дом «Русская редакция», 2002 – 480 с.

71. Network Protocols Handbook. – Javvin Technologies, 2005 -340 с.

72. Спецификации SOAP 1.2 (http://www.w3.org/TR/soap/).

73. **Ньюкомер Э.** Веб-сервисы. XML, WSDL, SOAP и UDDI/ Ньюкомер Э.–СПб.: Питер, 2003 – 256 с.

74. **Saint-Andre P.** XMPP. The Definitive Guide// Saint-Andre P., Smitb K., Troncon R. –Sebastopol: O'Relly, 2009.

75. **Bogdanov A.** Unified Memory Space Protocol Specification/ Bogdanov A., RFC 3018, December 2000.

76. **Олифер Н.А.** Сетевые операционные системы: учебник для ВУЗов/ Олифер Н.А., Олифер В.Г., – Питер, 2009 – 672 с.

77. Офіційний сайт Gigabyte (http://www.gigabyte.ru).

78. Краш-тест відеокарт (http://comprad.narod.ru/Hardtrable/videocmp.html).

79. **Ершова Н.Ю.** Микропроцессоры: электронное учебное пособие/ Ершова Н.Ю., Ивашенков О.Н., Курсков С.Ю., (http://dfe3300.karelia.ru/koi/posob/microcpu/index.html).

80. **John Sharp.** Microsoft Windows Communication Foundation Step by Step/ John Sharp, – Microsoft Press 2007- 448c., ISBN:9780735623361

81. **Craig McMurtry**. Microsoft Windows Communication Foundation: Hands-on/ Craig McMurtry, Marc Mercuri, – Sams, 2006.

82. **Chris Peiris**. Pro WCF: Practical Microsoft SOA Implementation / **Chris Peiris**, Dennis Mulder, – Microsoft Press, 2007 - 206 с., ISBN-13 (pbk): 978-1-59059-702-6.

83. **Justin Smith.** Inside Windows Communication Foundation / Justin Smith, – N-Y.: Microsoft Press, 2006, ISBN 9780735623064.

84. **Федоров А.** Windows Azure. Облачная платформа Microsoft/ Федоров А., Мартынов Д., – N-Y.: Microsoft Press, 2010, 100 с.

85. **Фаулер**. Архитектура корпоративных программных приложений/ Фаулер, Мартин, — М.: Издательский дом "Вильяме", 2006. — 544 с.

86. **Я. Фостер.** Анатомия ГРИД. Создание Масштабируемых виртуальных организаций/ **Я. Фостер**, К. Кессельман, С. Тьюк, - 2003.

87. **Я. Фостер**. Физиология ГРИД. Открытая архитектура грид-служб для интеграции распределённых систем/ Я. Фостер, К. Кессельман, Д Ник, С. Тьюк, - 2003.

88. Apache Hadoop. [Електроний ресурс] http://hadoop.apache.org/

89. The Hadoop Distributed File System: Architecture and Design. [Електроний ресурс] http://hadoop.apache.org/core/docs/current/hdfs_design.html

90. Microsoft Dryad. [Електроний ресурс] http://research.microsoft.com/research/sv/dryad/

91. **Michael Isard.** Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks./ Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, Dennis Fetterly, =- European Conference on Computer Systems (EuroSys), Lisbon, Portugal, March 21-23, 2007.

92. **Anthony Velte**. Cloud Computing: a practical approach. / Anthony Velte, Toby Velte, Robert Elsenpeter, 2010.

93. Amazon Elastic Compute Cloud [Електронний ресурс] http://aws.amazon.com/ec2/

94. Amazon Simple Storage Service [Електронний ресурс] http://aws.amazon.com/s3/

95. What Is Google App Engine? [Електронний ресурс] http://code.google.com/appengine/docs/whatisgoogleappengine.html

96. Google App Engine Developer's Guide: Python [Електронний ресурс] http://code.google.com/appengine/docs/python/overview.html

97. Google Apps [Електронний ресурс] www.google.com/apps/

98. Eucalyptus documentation [Електронний ресурс] http://open.eucalyptus.com/wiki/Documentation