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 2: «Несуперечливість і реплікація», «Відмовостійкість», «Захист інформації», «Розподілені системи об’єктів», «Розподілені файлові системи», «Розподілені системи документів», «Розподілені системи узгодження», «Пошукові системи»)

для студентів спеціальностей
8.092401
«Телекомунікаційні системи та мережі»

8.092402
«Інформаційні мережі зв’язку»

Київ – 2013

АНОТАЦІЯ

Підручник «Розробка інформаційних ресурсів та систем» призначений для допомоги студентам в засвоєнні методологічних основ побудови розподілених інформаційно-телекомунікаційних систем та середовищ, в тому числі й Internet – базованих, вивченні принципів утворення інформаційно-обчислювального середовища корпоративних систем прикладного призначення, а також підходів до просування інформаційних ресурсів компаній в пошукових машинах. Розглянуті базові складові, що визначають поняття інформаційно-обчислювального середовища, сучасні архітектурні рішення та технології щодо його створення та ефективного використання обчислювальних сервісів та інформаційних ресурсів, в тому числі й технології підтримки паралельних та «хмарних» обчислень.

Підручник складається з трьох томів. Перший том містить наступні розділи: «Розподілені системи», «Розподілені системи. Поняття розподіленого середовища», «Зв’язок», «Процеси», «Іменування», «Синхронизація». Другий том включає: «Несуперечливість і реплікація», «Відмовостійкість», «Захист», «Розподілені системи об’єктів», «Розподілені файлові системи», «Розподілені системи документів», «Розподілені системи узгодження», «Пошукові системи». Третій том присвячений вивченню архітектури паралельних систем, принципам організації паралельних алгоритмів, моделям та методам паралельних обчислень, а також підходам, принципам побудови і технологіям «хмарних» обчислень та Grid-систем.

Курс «Розробка інформаційних ресурсів та систем» входить до циклу професійної та практичної підготовки. Підручник розраховано на студентів-магістрів спеціальностей 8.092401 «Телекомунікаційні системи та мережі» та 8.092402 «Інформаційні мережі зв’язку», які

засвоїли курси інформатики, основи теорії телекомунікаційних мереж, алгоритмічне та програмне забезпечення телекомунікаційних мереж, захист інформації в телекомунікаційних мережах, бази даних, програмні технології корпоративних мереж та систем, системи адміністрування ТКС, а також на аспірантів та інженерів відповідної галузі знань.

Особлива увага у підручнику приділена вивченню методів та інструментальних засобів розробки інформаційних ресурсів корпоративних інформаційно-телекомунікаційних систем та мереж, сучасним практичним рішенням та технологіям в області проектування, реалізації та супроводу розподілених інформаційних систем, зокрема акцент зроблено на архітектурних рішеннях і технологіях щодо Internet – базованих систем, а також на розгляді технологій створення інформаційних ресурсів, які розміщуються в глобальному телекомунікаційному середовищі, використовують у якості технологій доступу Intranet-технології; на методах організації та взаємодії розподілених інформаційних та апаратних ресурсів таких систем, тощо.

Курс «Розробка інформаційних ресурсів та систем» ґрунтується на знаннях в області математичних методів та інформаційних технологій, змістом його є:

- технології створення та використання інформаційних та програмних ресурсів;
- основні характеристики територіально – розподілених систем та середовищ, в тому числі Internet – базованих;
- технології створення та експлуатації територіально – розподілених систем та середовищ;
- методи, алгоритми, технології та протоколи організації зв'язку між компонентами таких систем;
- організація обчислювальних процесів, використання системи імен;

- забезпечення синхронізації в системах, підтримка цілісності інформаційних ресурсів, непротирічності та реплікацій, відмовостійкості;
- забезпечення захисту інформаційних та програмних ресурсів розподілених систем;
- використання розподіленої системи об’єктів, розподіленої файлової системи, розподіленої системи документів та розподіленої системи узгоджень;
- підтримка розробки розподілених середовищ колективної роботи.

Теоретичний матеріал підручника відповідає навчальній програмі, написаній з використанням сучасних знань, яка спирається на наукові та практичні досягнення.

ЗМІСТ

ПЕРЕДМОВА

Підручник «Розробка інформаційних ресурсів та систем» призначено для студентів-магістрів, аспірантів та інженерів 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, розглянуто принципи утворення інформаційно-обчислювального середовища корпоративних систем прикладного призначення, а також підходи до просування інформаційних ресурсів компаній у пошукових машинах. Збільшення обсягів інформатизації в усіх галузях життєдіяльності людини й суспільства потребує підготовки саме таких спеціалістів, здатних до самостійного опанування різноманітних аспектів роботи в галузі телекомунікаційних та комп`ютерних технологій, коли необхідно у стислі терміни ефективно опрацьовувати значну кількість інформації.

Системи розподіленої обробки даних в Intranet-мережі належать до найбільш прогресивних форм організації програмно-технічних засобів у вигляді продуктивного інформаційно-телекомунікаційного середовища, використовують сучасні технології паралельних і «хмарних» обчислень, Grid-технології. Їх проектування і реалізація вимагають умов роботи, за яких користувачі повинні мати доступ до всіх файлів, які зберігаються у вузлах інформаційно-телекомунікаційної мережі. Ефективність доступу користувачів значною мірою залежить від організації інформаційно-обчислювального середовища, зокрема з використанням мережі Internet.

Оскільки здебільшого одні корпоративні інформаційно-телекомунікаційні системи відрізняються від інших як апаратним, так і

програмним забезпеченням, то організувати колективну роботу користувачів у оперативному режимі досить складно через суттєву неструктурованість інформації. Виходячи із цього, слід зазначити, що оптимізація ефективності роботи корпоративних систем стає нагальною проблемою, вирішення якої можливе, якщо одночасно використовувати різні сучасні технології зв'язку та інформаційних технології.

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 інформа-ційно-телекомунікаційних технологій та систем, його було поширено в роботах багатьох науковців [3, 6, 9, 12, 13, 14, 23, 30, 34, 35, 78, 82-100]. Виходячи з того, що багато сучасних наукових підходів для створення ефективних глобальних систем та технологій [17, 21, 28, 61, 71, 74] 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.

Підручник складається з двох томів. Другий том містить вісім розділів, у яких, крім достатньо повного висвітлення основ Intranet-технологій, ґрунтовно розглянуто телекомунікаційні та інформаційні технології, які підтримують роботу в глобальному середовищі, приклади сучасних Internet-базованих систем, а також методи просування сайтів у пошукових машинах, сервіси публікації\підписки на інформаційні та обчислювальні ресурси.

Відповідно розглянуто принципи створення, організації доступу та збереження інформаційних та програмних ресурсів у інфраструктурі

територіально-розподілених систем та середовищ, архітектурні рішення, які визначають ефективність їх використання. Забезпечення цілісності й несуперечливості інформації, вдосконалення механізмів її реплікації та відмовостійкості інформаційно-телекомунікаційної інфраструктури, забезпечення захисту інформаційно-обчислювальних ресурсів, розмежування прав доступу до них – основні питання, які висвітлено у підручнику.

Подальше, більш глибоке, проникнення в усі сфери діяльності людини інформаційно-телекомунікаційних технологій вимагає значного підвищення ефективності процесів інформаційного обміну, що забезпечується як за рахунок удосконалення апаратної інфраструктури, каналів зв'язку, протоколів 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 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, утворення 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 такі складні розподілені у глобальній мережі системи і кваліфіковано формулювати завдання на їх розробку.

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

Матеріал підручника, у якому розглянуто питання проектування та ефективного функціонування сучасних розподілених в Intranet-мережі систем, відповідає задачам і вимогам, обумовленим якісно новим рівнем інформатизації суспільства, у зв`язку з чим значно збільшено розділи, присвячені програмним технологіям побудови Intranet-мереж.

Зміст підручника ґрунтується на досвіді викладання відповідних курсів на кафедрі інформаційно-телекомунікаційних мереж Інституту телекомунікаційних систем 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» професору М.Ю. Ільченко, рецензентам підручника декану механіко-математичного факультету Харківського Національного університету ім. В. Н. Каразіна, завідувачу кафедрою 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, Л.С. Глобі.

1. НЕСУПЕРЕЧЛИВІСТЬ І РЕПЛІКАЦІЯ

1.1. Значення реплікації

Реплікація даних є важливим чинним підвищення ефективності функціонування розподілених систем. Дані зазвичай реплікуються для *підвищення надійності* та *збільшення продуктивності*. Одна з основних проблем при цьому - **збереження несуперечливості реплік**. Якщо в одну з копій вносяться зміни, то необхідно забезпечити внесення цих змін у інші копії, інакше репліки не будуть однаковими.

Сучасні інформаційні системи висувають досить жорсткі вимоги до швидкості опрацювання інформації за умови одночасної роботи великої кількості клієнтів. Окрім того, розвиваючись, такі системи мають легко масштабуватися, не погіршуючи швидкісних характеристик системи. Один із способів задоволення цієї потреби - створення розподіленої бази даних (далі – БД), яка підтримує механізм асинхронної реплікації даних, коли замість однієї БД, з якою повинні працювати всі клієнти інформаційної системи, створюється кілька однакових серверів БД на різних машинах і/або вузлах мережі. Клієнти мають доступ до деякого розподільного пристрою (реалізованого апаратно або програмним методом), який у разі підключення нового клієнта оцінює завантаження кожного сервера БД і спрямовує клієнта до найменш завантаженого сервера, з яким клієнт і буде працювати до роз'єднання.

Питання побудови розподіленої БД єдиної інформаційної системи виникають й у процесі розвитку компаній, коли створюють віддалені філії, магазини та склади. Кожна віддалена інформаційна система для підвищення відмовостійкості має працювати самостійно, періодично відправляючи до Центрального офісу консолідовану інформацію. Для виключення негативного впливу людського фактора під час

періодичної синхронізації інформації БД слід включати до загальної системи реплікації.

Реплікація даних між серверами баз даних може виконуватися за допомогою вбудованих засобів системи керування базами даних (СКБД) або реалізуватися у межах бізнес-логіки прикладного програмного забезпечення. Реплікація за допомогою вбудованих засобів СКБД передбачає наявність надійних каналів зв'язку. Пропускна здатність цих каналів має бути достатньо високою, щоб встигати передавати всю інформацію, яка реплікується в режимі реального часу. В основі процесу реплікації в СКБД лежать поняття «видавець», «передплатник», «стаття». Видавцем є сервер публікації, тобто сервер, який надсилає інформацію, а передплатником - відповідно сервер, який приймає інформацію (сервер передплати).

Налаштування реплікації полягає в установленні відносин між видавцем і передплатником. Недоліком такої реплікації є однонапрямність, тобто стаття (фактично – таблиця) може передаватися від видавця передплатнику, при цьому передплатник не може її змінювати.

Реалізація процесів реплікації на рівні бізнес-логіки значно ускладнює розробку прикладних систем, але дозволяє оптимізувати сам процес передачі інформації. Реплікація інформації є досить нетривіальну задачею з неоднозначним рішенням. Розпочинаючи розв’язання задачі реплікації даних, необхідно враховувати наявність конфліктів даних, які реплікуються. Таких конфліктів для баз даних, які працюють у єдиній мережі з використанням прямого зв’язку із сервером бази даних, не виникає. Особливо складним є перехід від єдиної бази до розподіленої, коли доводиться підлаштовувати алгоритм реплікації під вже наявну структуру, в якій працює БД. Розробляючи нову інформаційну систему потрібно враховувати технологічні нюанси майбутньої розподіленої бази даних.

Щоб досягти високої продуктивності в операціях зі спільно використовуваними даними, розробники паралельних комп'ютерів приділяють значну увагу різним *моделям несуперечливості даних* у розподілених системах з *розподілюваною пам'яттю*. Ці моделі успішно застосовують для різних типів розподілених систем.

Реалізація **моделей несуперечливості розподілюваної пам'яті** для великомасштабних розподілених систем зазвичай є нелегким завданням. Однак часто можна використати прості моделі, які нескладно реалізувати. Один із конкретних прикладів таких моделей – *клієнтські моделі несуперечливості*, які обмежуються несуперечливістю з погляду одного (можливо, мобільного) клієнта.

Для підтримки несуперечливості реплік використовують два незалежні принципи: перший - це **поширення оновлень,** тобто розміщення реплік і способів поширення оновлення ними; другий - підтримка **несуперечливості реплік**. Здебільшого програмам необхідна строга несуперечливість. Оновлення мають поширюватися репліками швидко, крім того важливими є протоколи кешування, які є особливою формою *протоколів підтримки несуперечливості*.

Функціональні вимоги до сервера реплікації. До розробки сервера реплікації висуваються такі основні вимоги:

- *поєднання функції* сервера публікації і сервера передплати;
- можливість використання *в гетерогенній (змішаній)* системі, тобто система реплікації має охоплювати як Oracle, так і MS SQL, або інші сервери баз даних;
- *багатонапрямна реплікація з гарантованою доставкою* інформації, тобто сервер публікації має розсилати інформацію одного сервера передплати;
- *сервер підписки* повинен мати можливість приймати інформацію від *декількох серверів публікації*;

- система реплікації може являти собою складну павутину, в якій окремі сервери реплікації, поєднуючи функції серверів передплати й публікації, виконують функцію *сервера пересилання інформації*;
- наочний *візуальний контроль* функціонування сервера реплікації як у режимі публікації, так і в режимі передплати;
- реплікація інформації таблиць, до структури яких входять поля BLOB (binary large object) та поля, які допускають використання значення NULL;
- *кодування* інформації, яка реплікується.

Сервер реплікації має задовольняти не тільки ці вимоги, але й виконувати додаткові функції, які дозволяють наочно контролювати процес реплікації, встановлювати і контролювати параметри каналу з'єднання з віддаленими серверами, виконувати експортно-імпортні операції для доставки інформації на жорстких носіях та завантаження її в сервер підписки, якщо немає каналу зв'язку.

Сервер ліцензії, що входить у комплектацію сервера реплікації, забезпечує додатковий захист інформації, яка реплікується. Система реплікації замкнена, і сервер ліцензії не дозволяє серверу підписки під'єднуватися до сервера публікації іншої інформаційної системи, а також відмовляє в доступі серверам публікації інших інформаційних систем [56].

1.1.1. Необхідність реплікації

Використання реплікації дозволяє отримати дві основні переваги - **надійність і продуктивність**.

1. Дані реплікуються для підвищення надійності системи, оскільки файлова система реплікована, то вона може продовжувати свою роботу

після збою в одній із реплік, просто переходячи на іншу. Підтримуючи кілька копій, легше протистояти збоям даних.

Приклад. Існує три копії якогось файлу, причому операції читання і запису здійснюються з усіма трьома файлами одночасно. Можна захиститися від одиничної невдалої операції запису, вважаючи правильними значення, які отримано як мінімум з двох копій.

2. Реплікація підвищує продуктивність, коли розподілену систему доводиться масштабувати на велику кількість машин і географічних зон, зокрема коли зростає кількість процесів, які вимагають доступу до даних, керованих одним сервером. За таких умов продуктивність можна підвищити за допомогою реплікації сервера з подальшим поділом загального обсягу роботи між сервером і репліками.

Реплікацію можна *класифікувати* по-різному. Розглянемо такі варіанти:

1. *За спрямуванням реплікації:*

- однонапрямлена або однобічна реплікація – дані змінюються тільки в одній з БД;
- багатонапрямлена або багатобічна реплікація – дані можуть змінюватися і вводитися в усіх БД.

2. *За часом проведення сеансу* реплікації:

- реплікація реального часу – дані мають бути синхронізованими негайно після змін;
- відкладена або асинхронна реплікація – процес реплікації запускається за якою-небудь подією в часі.

3. *За способом передачі інформації* у процесі реплікації:

- *пряма реплікація* – виконується за допомогою програми клієнта, яка має стійке з'єднання із серверами розподілених БД;
- *недетермінована або ймовірнісна реплікація* – виконується за допомогою програми клієнта, яка не має стійкого з'єднання із серверами розподілених БД.

4. *За способом аналізу* інформації, яка підлягає реплікації:

- *реплікація станом* – алгоритм реплікації працює за принципом порівняння записів однієї таблиці з записами другої, на підставі чого приймається рішення про синхронізацію;
- *реплікація змін* – алгоритм реплікації переносить до реплік лише зміни інформації в БД, які зберігаються в журналі внесених транзакцій БД [56].

Для масштабування на географічну зону, що збільшилася, також необхідна реплікація. Основна ідея реплікації у такому разі полягає в переміщенні копії даних ближче до процесу, який ними користується, що зумовлює зменшення часу доступу, в результаті чого підвищується продуктивність процесу. Це означає і те, що виграш у продуктивності, одержуваний завдяки реплікації, оцінити не просто. Незважаючи на підвищення продуктивності у клієнтському процесі, можлива ситуація, у якій для своєчасного оновлення всіх реплік потрібно збільшення пропускної здатності мережі.

Реплікація допомагає підвищити надійність і продуктивність, але також спричиняє певні проблеми, зокрема *з несуперечливістю даних,* через наявність множини копій, адже щоразу зі змінюванням копії вона відрізнятиметься від усіх інших. Отже, для збереження несуперечливості ці зміни мають бути перенесеними й на інші копії, а порядок реплікації, тобто як і коли варто переносити ці зміни, визначає **ціну реплікації**.

Приклад. Розглянемо проблему постійного зростання часу доступу до web-сторінок. Якщо не вживати ніяких спеціальних заходів, то завантаження сторінки з віддаленого web-сервера може тривати до декількох секунд. Для підвищення продуктивності web-браузери часто локально зберігають копії завантажених раніше web-сторінок (тобто кешують їх). Якщо користувач знову запросить таку сторінку, то браузер автоматично поверне йому локальну копію. Швидкість доступу, з погляду користувача, буде досить високою, однак можливість мати останню версію сторінки для користувача обмежена. Проблема полягає в тому, що

коли у проміжку між зверненнями сторінка стане модифікованою, модифікації не поширюватимуться на копії в кеші і копії будуть застарілими.

Одне з вирішень проблеми повернення користувачеві актуальних копій полягає в тому, щоб заборонити браузеру зберігати локальні копії, а завдання оновлення повністю покласти на сервер, однак це може призвести до збільшення часу доступу, адже тоді у користувача не буде реплік. Друге вирішення – дозволити web-серверу оголосити кешовані копії застарілими або оновлювати їх, але тоді серверу доведеться перевіряти всі кешовані копії та розсилати їм оголошення. Це, у свою чергу, може зумовити зниження загальної продуктивності сервера.

1.1.2. Реплікація об'єктів

Щоб краще зрозуміти роль розподілених систем у керуванні даними (спільно використовуваними), варто розглядати не окремі дані, а об'єкти, перевага яких полягає в тому, що вони інкапсулюють дані та операції над ними. Це полегшує поділ операцій на операції, які залежать від даних, і операції, які від даних не залежать.

Розглянемо розподілений віддалений об'єкт, спільно використовуваний багатьма клієнтами (рис. 1.1). Перед тим, як розпочинати його реплікацію на кілька машин, потрібно вирішити проблему захисту об'єкта від одночасного доступу декількох клієнтів. У цієї проблеми наявні два основні розв'язки.

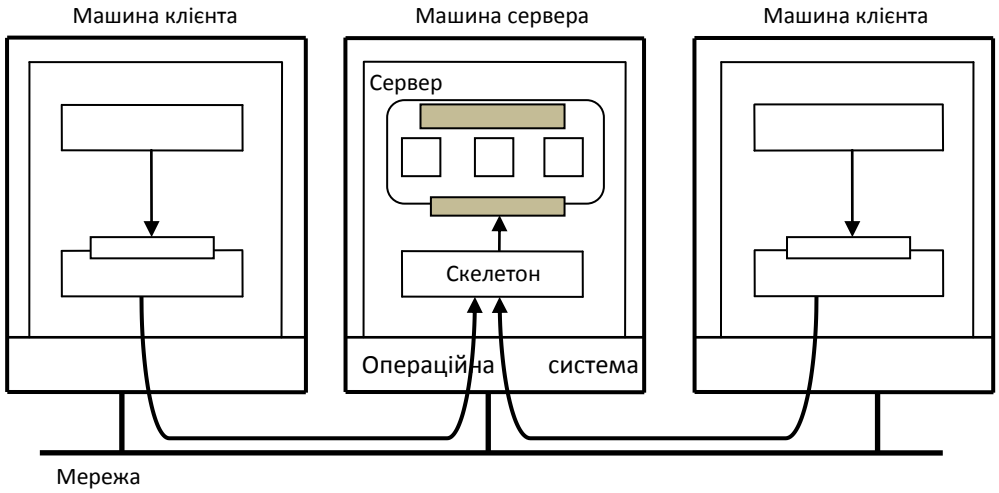


Рис. 1.1. Організація розподіленого віддаленого об'єкта, спільно використовуваного двома клієнтами

Перший розв'язок полягає у тому, що одночасні виклики обробляє сам об'єкт.

Приклад. Об'єкт Java може бути сконструйованим у вигляді монітора за рахунок оголошення його методів **синхронізованими**. Припустімо, два клієнти одночасно викликають метод одного об'єкта. Це зумовить появу двох паралельних потоків виконання на тому сервері, на якому перебуває цей об'єкт. У Java, якщо методи об'єкта синхронізовані, виконуватиметься тільки один з потоків виконання, другий буде заблокованим до подальших повідомлень. Можуть бути створені різні рівні паралельного виконання, але основним моментом буде те, що засоби обробки паралельних запитів реалізує сам об'єкт. Цей принцип ілюструє рис. 1.2, *а*.

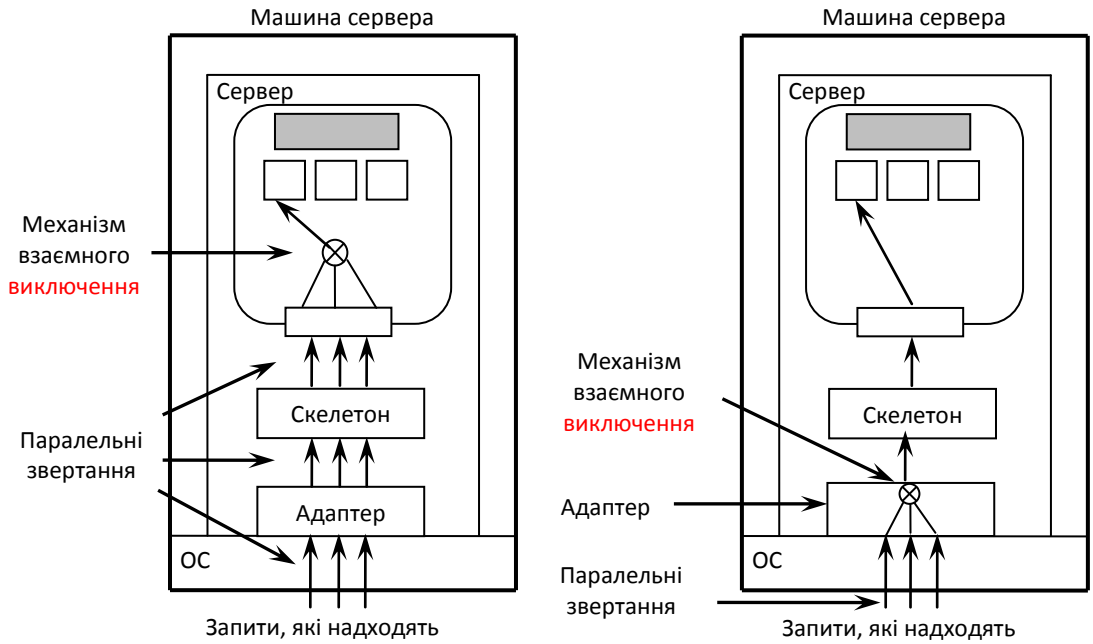


Рис. 1.2. Віддалений об'єкт, здатний самостійно обробляти паралельні звертання (*а*); віддалений об'єкт, якому для обробки паралельних запитів необхідний адаптер об'єкта (*б*)

Відповідно до **другого розв'язку** об'єкт взагалі ніяк не захищається від паралельних запитів, замість нього відповідальність за роботу із цими запитами несе сервер, на якому розміщений об'єкт. Зокрема, використовуючи відповідний адаптер об'єктів, він може забезпечити, щоб паралельні запити не ушкоджували об'єкт.

Приклад. Адаптер об'єктів, що використовує для кожного об'єкта один потік виконання, може серіалізувати весь доступ до кожного з об'єктів, якими він керує (рис. 1.2, *б*).

Реплікація віддалених, спільно використовуваних об'єктів без спеціальних прийомів обробки паралельних запитів може призвести **до проблем з несуперечливістю**. Ці проблеми пов'язані з тим, що для паралельних запитів у правильній черговості репліки потребують в **додаткової синхронізації.** Для вирішення цієї проблеми використовують два основних підходи.

Перший підхід грунтується на поінформованості об'єкта про те, що його було репліковано, тоді сам об'єкт відповідає за несуперечливість своєї репліки в умовах паралельних запитів. Такий підхід є подібним до до моделі об'єктів, які самостійно обробляють паралельні запити. Розподілені системи, які пропонують подібні об'єкти, зазвичай не мають потреби в централізованій підтримці реплікації. Підтримка може бути обмежена наданням серверів та адаптерів, які допомагають у побудові обізнаних про реплікацію об'єктів (рис. 1.3, *а*), перевага таких об'єктів полягає в тому, що вони можуть реалізувати специфічну для об'єкта стратегію реплікації, подібно до того, як об'єкти, що працюють паралельно, можуть обробляти паралельні запити будь-яким особливим способом.

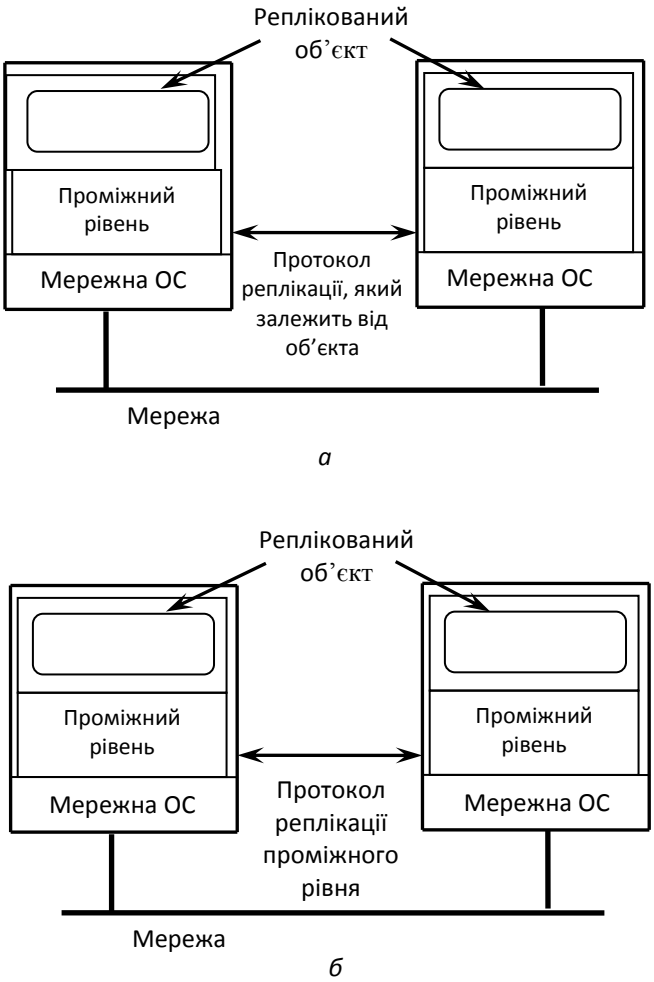


Рис. 1.3. Розподілена система для обізнаних про реплікацію розподілених об'єктів (*а*); розподілена система, яка відповідає за керування репліками (*б*)

Другий, більш загальний, підхід до забезпечення несуперечливості паралельних об'єктів – зробити відповідальною за керування реплікацією розподілену систему, як це показано на рис. 1.3, *б*, зокрема, за те, щоб паралельні запити перенапрямлялися різним реплікам у правильному порядку. Використання розподіленої системи для керування реплікацією дозволяє спростити розробку програмного забезпечення, при цьому іноді буває нелегко адаптувати специфічні для об'єктів рішення, що є недоліком таких систем, але ці рішення часто потрібні для масштабування.

1.1.3. Реплікація як метод масштабування

Методи реплікації та кешування, які підвищують продуктивність, часто використовують і як метод масштабування, який зумовлює зниження продуктивності. Однак розміщення копій даних та об'єктів неподалік від процесів, які їх використовують, завдяки зменшенню часу доступу дозволяє підвищити продуктивність. Разом із тим, виникає необхідність збереження актуальності копій, що потребує додаткової пропускної здатності мережі.

Приклад. Розглянемо процес *P*, який звертається до локальної репліки *N* разів за секунду, у той час як сама репліка оновлюється *M* разів за секунду. Припустімо, оновлення повністю змінює попередню версію локальної репліки. Якщо $N<<M$, тобто співвідношення звертань до оновлень дуже низьке, виникає ситуація, коли запитів до багатьох оновлених версій з боку *P* просто не буде і використовувати мережний зв'язок для поширення цих версій не треба. У цьому разі краще не встановлювати локальну репліку поблизу процесу *P* або застосувати іншу стратегію оновлення цієї репліки.

Для коректної роботи розподіленої системи у разі її масштабування необхідно зберігати актуальність багатьох копій, які є актуальними, якщо всі копії постійно однакові, тобто операція читання має давати однакові результати для кожної з копій. Відповідно, у разі виконання операції оновлення однієї з копій оновлення має поширитися на всі копії до того, як почнеться наступна операція, при цьому не має значення, з якої копії почалася ця операція. Такий тип несуперечливості іноді неправильно називають **щільною несуперечливістю**, оскільки вона підтримує так звану **синхронну реплікацію**. Ключовою є ідея про те, що оновлення всіх копій має відбуватись як одна атомарна операція або транзакція. На жаль, реалізація атомарності операції, якщо до неї залучено велику кількість реплік, які, до того ж, можуть бути розкиданими по вузлах великої

мережі, суттєво ускладнена, особливо коли ця операція також має бути виконана за короткий час.

Труднощі виникають через те, що необхідно синхронізувати всі репліки, тобто репліки мають узгодити точний час, коли відбуватиметься їх локальне оновлення. Наприклад, у реплік може виникнути потреба в узгодженні глобального впорядкування операцій, використовуючи механізм оцінювання часу Лампорта або координатора, який диктуватиме їм порядок дій. Глобальна синхронізація забере багато часу на взаємодію, особливо якщо репліки розкидані по глобальній мережі.

Отже, з одного боку, гостроту проблем масштабованості можна знизити, використовуючи реплікацію та кешування, які забезпечують підвищення продуктивності; з другого, щоб підтримувати всі копії в актуальному стані, зазвичай потрібна глобальна синхронізація, що суттєво впливає на продуктивність.

У багатьох випадках єдине реальне рішення полягає у пом'якшенні обмежень, тобто, якщо зняти вимогу про атомарність операцій оновлення, відпаде потреба у необхідності миттєвої глобальної синхронізації, що дозволить підвищити продуктивність, але у такому разі з'являється можливість появи неоднаких копій. Той рівень, до якого можливо зменшити вимоги несуперечливості, залежить від варіантів доступу до реплікованих даних і способів їх оновлення, а також від завдань, у яких використовуються ці дані.

1.2. Моделі несуперечливості, орієнтовані на дані

Традиційно несуперечливість розглядають у контексті операцій читання і запису над сумісно використовуваними даними, доступними в розподіленій пам'яті (поділюваній) або у файловій системі (розподіленій). Введемо загальний термін «**сховище даних»** (data

store). Сховище даних може бути фізично рознесене по декількох машинах. Зокрема, кожний з процесів, які бажають одержати доступ до даних зі сховища, може означати наявність доступної через сховище локальної (або розміщеної неподалік від нього) копії даних. Операції запису поширюються й на інші копії, як це показано на рис. 1.4. Операції над даними називають **операціями запису**, якщо вони **змінюють дані,** решту операцій уважають **операціями читання**.

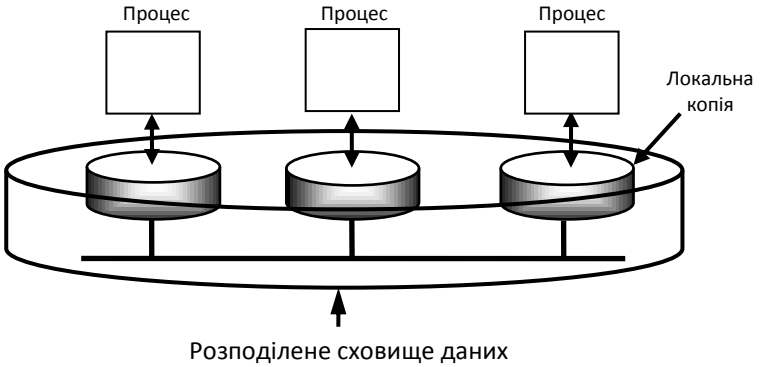


Рис. 1.4. Узагальнена організація логічного сховища даних, фізично розподіленого і реплікованого за декількома процесами

Модель несуперечливості (consistency model) - це певний контракт між процесами і сховищем даних, який передбачає, що, якщо процеси згодні дотримуватися деяких правил, то сховище погоджується працювати правильно**.** Зазвичай процес, який виконує операцію читання елемента даних, очікує, поки операція поверне значення, яке відповідає результату останньої операції запису цих даних.

Якщо немає глобального годинника, то важко точно визначити, яка з операцій запису була останньою, тому потрібно ввести інші, альтернативні визначення, які допоможуть створити моделі несуперечливості. Кожна модель успішно обмежуватиме набір значень, які може повернути операція читання елементу даних.

Кількість обмежень, які накладаються на операції читання\запису елементів даних може бути значною, що впливає на складність процесу їх реалізації. Моделі з мінімальним обсягом обмежень реалізувати

простіше на відміну від моделей з максимальними обмеженнями, але прості моделі працюють гірше, ніж складні.

1.2.1. Строга несуперечливість

Найбільш жорстку модель несуперечливості називають **строгою несуперечливістю** (strict consistency). Вона визначається такою умовою: будь-яке читання елемента даних *x* повертає значення, що відповідає результату останнього запису *x*.

Це визначення цілком зрозуміле, хоча передбачає існування абсолютного глобального часу (як у класичній фізиці), коли визначення останньої події є однозначним, бо вона характеризується найбільшим часом. Однопроцесорні системи традиційно дотримуються строгої несуперечливості.

Приклад. Розглянемо таку програму: ***a = 1; a = 2; print(a);***

Система, у якій ця програма надрукує *1* або будь-яке інше значення, крім *2*, швидко призведе неправильного виконання.

Для системи, у якій дані розкидані по декількох машинах, а доступ до них має кілька процесів, усе значно ускладнюється. Допустімо, *x* - це елемент даних, що зберігається на машині *B*. Уявімо, що процес, який працює на машині *A*, читає *x* у момент часу T_1, тобто посилає *B* повідомлення з вимогою повернути *x*. Трохи пізніше, у момент часу T_2, процес із машини *B* робить запис *x*. Якщо строга несуперечливість зберігається, читання має завжди повертати попереднє значення, яке не залежить від того, де перебувають машини і наскільки малий інтервал між T_1 і T_2. Однак якщо T_2-T_1 дорівнює, скажімо, одній наносекунді, а машини розташовані у трьох метрах одна від одної, то щоб запит на читання від *A* дійшов до машини *B* раніше надсилання машиною *B* запиту на запис, він має рухатися в 10 разів швидше від швидкості світла, що є фізично неможливим.

Проблема зі строгою несуперечливістю полягає в тому, що вона потребує знання абсолютного глобального часу. По суті, у розподіленій системі неможливо встановити на кожну операцію унікальну відмітку часу, погоджену з дійсним глобальним часом. Для того, щоб не

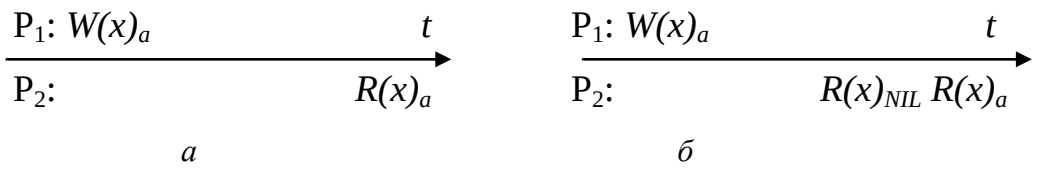
погоджувати унікальну відмітку часу кожної операції з дійсним глобальним часом час розділяють на серії послідовних інтервалів, які не перекриваються. Кожна операція може відбуватися в межах інтервалу, ій призначають відмітку часу, яка відповідає цьому інтервалу. Залежно від того, наскільки точно можна синхронізувати годинники, може виникнути ситуація, коли на один такий інтервал припаде більше однієї операції.

Таким чином, вірогідність того, що на один інтервал буде припадати максимум одна операція, дуже мала, відповідно, необхідно враховувати можливість наявності в одному інтервалі декількох операцій. Аналогічно до ситуації паралельного виконання розподілених транзакцій, виконання двох операцій у тому самому інтервалі призводить до конфлікту, якщо вони проводяться над одними й тими ж даними, і одна з них - операція читання. Важливим для визначення моделі несуперечливості є точне з'ясування того, який тип поводження застосовувати при наявності конфліктуючих операцій.

Приклад. Для детального вивчення несуперечливості наведемо кілька прикладів. Щоб ці приклади були точнішими, треба навести спеціальні нотації, за допомогою яких відобразимо операції процесів на осі часу. Вісь часу завжди рисують горизонтально, час збільшується зліва направо. Символи $Wi(x)_a$ і $Ri(x)_b$ означають відповідно, що виконано запис процесом P в елемент даних x значення a і читання із цього елемента процесом P, який повернув b. Уважають, що кожен елемент даних спочатку був ініціалізований нулем. Якщо під час процесу доступу до даних не було ніяких збоїв, то індекси a, b символів W і R опустимо.

Як приклад розглянемо рис. 1.5, *а*. Процес P_1 здійснює запис в елемент даних x, змінюючи його значення на a. Відзначимо, що операція $W_1(x)_a$, спочатку копіює значення в локальне сховище даних P_1, а потім поширює його й на інші локальні копії. У нашому прикладі P_2 пізніше читає значення x з його локальної копії у сховищі й виявляє там значення a. Таке поводження характерно для сховища даних, яке зберігає строгу несуперечливість. Натомість процес P_2 (рис. 1.5, *б*) виконує читання після запису (можливо, лише на наносекунду пізніше, але пізніше)

та одержує нуль (*NULL*). Наступне читання повертає *a*. Таке поводження для сховища даних, яке зберігає строгу несуперечливість, некоректно.



а *б*

Рис. 1.5. Поводження двох процесів, які працюють із тим самим елементом даних: горизонтальна вісь — час, а сховище зі строгою несуперечливістю - (*а*), (*б*) - сховище без строгої несуперечності

Отже, коли сховище даних строго несуперечливе, всі операції запису миттєво помічають усі процеси. Витримується абсолютний глобальний порядок за часом. Якщо елемент даних змінюється, то всі подальші операції читання, які виконуються з даними, повертають нове значення, при цьому не важливо, як швидко після зміни виконується читання та який процес здійснює читання і де він відбувається. Відповідно, якщо виконується читання, то воно повертає це поточне значення, і не важливо, як швидко після цього виконується наступний запис.

1.2.2. Лінеаризуємість і послідовна несуперечливість

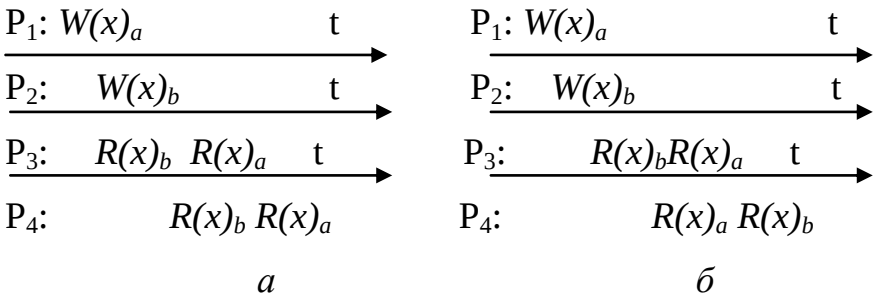
Хоча строга несуперечливість і є ідеальною моделлю несуперечливості, реалізувати її в розподілених системах неможливо. В операційних системах розглядають проблеми критичних областей і взаємних виключень, при цьому правильно написані паралельні програми не мають використовувати ніяких припущень про відносні швидкості процесів або про черговість виконання інструкцій. Обробка двох подій в одному процесі відбувається так швидко, що інший процес не в змозі усунути нівіть виявлене протиріччя, замість чого у процесі програмування не підтримується точний порядок виконання інструкцій (посилань на пам'ять). Якщо важливою є черговість подій, то потрібно використати **семафори** або інші операції синхронізації. Це означає використання **моделі слабкої несуперечливості**.

Послідовна несуперечливість (sequential consistency) – це менш строга модель несуперечливості. Вперше її було визначено в контексті

спільно використовуваної пам'яті мультипроцесорних систем. Загалом сховище даних уважають послідовно несуперечливим, якщо воно задовольняє такій умові: *результат будь-якої дії такий самий, коли операції (читання й запису) всіх процесів у сховище даних виконувалися у деякому послідовному порядку, причому операції кожного окремого процесу - у порядку, зумовленому його програмою.*

Це означає, що коли процеси виконуються паралельно на (можливо) різних машинах, будь-яке правильне чергування операцій читання й запису є допустимим, *але всі процеси бачать одне й те ж чергування операцій.* Відзначимо, що фактор часу виконання операції ніяк не впливає, тобто ніякий процес не посилається на останню операцію запису об'єкта, у цьому контексті процес «бачить» записи всіх процесів, але тільки тих, які сам читає.

Приклад. Розглянемо роботу чотирьох процесів без врахування часу виконання операцій з одним елементом даних *x* (рис. 1.6). На рис. 1.6, *а* спочатку процес P_1 здійснює запис $W(x)_a$ в елемент *x*, пізніше (за абсолютним часом) процес P_2 також здійснює запис, установлюючи значення *x* в *b*. Однак обидва процеси, P_3 і P_4, спочатку читають значення *b* і лише потім – значення *a*. Інакше кажучи, операція запису процесу P_2 виглядає такою, що відбувається раніше запису процесу P_1.



а *б*

Рис. 1.6. Сховище даних з послідовною несуперечливістю - *(а),*
сховище даних без послідовної несуперечливості - *(б)*

На рис. 1.6, *б* послідовна несуперечливість порушується, оскільки не всі процеси бачать однакове чергування операцій запису. Зокрема, для процесу P_3 справа виглядає так, що елемент даних спочатку приймає значення *b*, а потім – *a*. З іншого боку, P_4 вважає, що підсумкове значення елемента даних – *b*.

Моделлю несуперечливості, яка більш слабка, ніж строга несуперечливість, але більш сильна, ніж послідовна, є **лінеаризуємість** (linearizability). Відповідно до цієї моделі операції одержують відмітку часу, використовуючи глобальні годинники, які мають кінцеву точність. Такі годинники можуть бути реалізовані в розподіленій системі за умови, що процеси мають слабко синхронізований годинник. Нехай $ts_{OP}(x)$ – це відмітка часу, призначена операції *OP*, що працює з елементом даних *x,* причому *OP* – це операція читання *(R)* або запису *(W)*. Сховище даних називається лінеаризованим, якщо кожна операція має відмітку часу та виконується наступна умова: результат кожної дії є таким, наче всі операції (читання й запису) всіх процесів зі сховища даних функціонують у деякому послідовному порядку, причому операції кожного окремого процесу функціонують у послідовності, визначеній у їх програмах, і, крім того, якщо $ts_{OP1}(x)<ts_{OP2}(x)$, то операція $OP_1(x)$ у цій послідовності повинна передувати операції $OP_2(y)$.

Лінеаризоване сховище даних характеризується ще й послідовною несуперечливістю, але його відмінність полягає у тому, що при врахуванні впорядкованості до уваги беруться також установки синхронізованих годинників. На практиці лінеаризуємість використовується в першу чергу як механізм формального підтвердження коректності роботи паралельних алгоритмів. Додаткові обмеження, такі як збереження впорядкованості відповідно до відміток часу, робить витрати на реалізацію лінеаризації вищими, ніж на реалізацію послідовної несуперечливості.

Послідовна несуперечливість нагадує серіалізацію у транзакціях. Нагадаємо, що набір транзакцій, які виконуються паралельно, вважається серіалізуємим, якщо підсумковий результат може бути отриманий шляхом виконання цих транзакцій послідовно з певною черговістю. Основна різниця полягає в ступені деталізації: послідовна

несуперечливість визначається в поняттях операцій запису і читання, а серіалізуємість - у поняттях транзакцій, які поєднують ці операції в одне ціле.

1.2.3. Причинна несуперечливість

Модель причинної несуперечливості (causal consistency) є ослабленим варіантом послідовної несуперечливості, за якої проводиться розподіл між подіями, які потенційно володіють причинно-наслідковим зв'язком, і подіями, які ним не володіють. Якщо подія *B* викликана попередньою подією *A* або перебуває під її впливом, то причинно-наслідковий зв'язок вимагає, щоб спостерігалася спочатку подія *A,* а потім подія *B*.

Приклад. Розглянемо модель причинної несуперечливості у разі звернення до пам'яттi. Припустімо, що процес P_1 записує значення змінної *x*. Читання *x* і запис *y* будуть потенційно пов'язані із цим процесом причинно-наслідковим зв'язком, якщо обчислення *y* може залежати від значення *x,* яке прочитає процес P_2 (від значення, записаного процесом P_1). Якщо два процеси випадково одночасно записують значення двох різних змінних, вони не мають причинно-наслідкового зв'язку.

Якщо за операцією запису виконується операція читання, ці дві події потенційно мають причинно-наслідковий зв'язок. Оскільки, операція читання є пов'язаною з операцією запису, яка надає дані для операції читання, то їх вважають пов'язаними причинно-наслідковим зв'язком. Операції, які не пов'язані причинно-наслідковим зв'язком, називаються *паралельними (concurrent)*.

Для того, щоб сховище даних підтримувало причинну несуперечливість, воно повинне задовольняти такій умові: *операції запису, які потенційно зв'язані причинно-наслідковим зв'язком, повинні спостерігатися всіма процесами в однаковому порядку, а паралельні операції запису можуть спостерігатися на різних машинах у різному порядку.*

Приклад. На рис. 1.7. представлена послідовність подій, можлива для сховища із причинною несуперечливістю, але заборонена для сховищ зі строгою або послідовною несуперечливістю. Варто відзначити, що $W_2(x)b$ і $W_1(x)c$ – паралельні операції, і тому їх черговість для різних процесів не важлива.

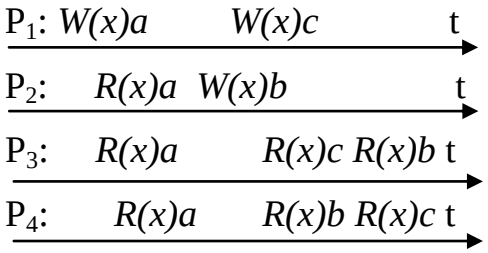


Рис. 1.7. Послідовність, допустима для сховища із причинною несуперечливістю

Приклад. На рис. 1.8, *а* видно, що $W_2(x)b$ потенційно залежить від $W_1(x)a$, оскільки *b* може бути результатом обчислень, які використовують значення, прочитане операцією $R_2(x)a$. Дві операції запису зв'язані причинно-наслідковим зв'язком, а тому всі процеси мають спостерігати їх в однаковому порядку. Таким чином, рис. 1.8, *а* некоректний. З іншої сторони, на рис. 1.8, *б* операція читання із процесу *P2* прибрана, і тепер *W1(x)a* й *W2(x)b* стали паралельними операціями запису. Сховище із причинною несуперечливістю не вимагає, щоб паралельні операції запису мали глобальну впорядкованість, і тому рис. 1.8, *б* коректний.

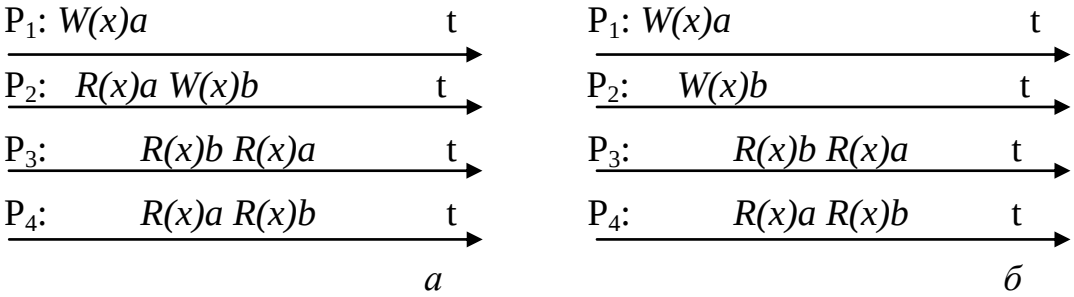


Рис. 1.8. Порушення причинної несуперечливості сховища - *(а),* правильна послідовність подій у сховищі з причинною несуперечливістю - *(б)*

Реалізація вимагає відслідковувати, які процеси які записи бачили. Для цього створюють та підтримують граф залежностей, на якому буде зазначено взаємну залежність операцій один від одного. Одним із способів позначення причинно-наслідкової залезності є використання векторних відміток часу.

1.2.4. Несуперечливість FIFO

У випадку причинно-наслідкової несуперечливості допустимо, щоб на різних машинах паралельні операції запису спостерігалися в різному порядку, але вимагається, щоб операції запису, зв'язані причинно-наслідковим зв'язком, мали однаковий порядок виконання, з якої б машини не велося спостереження. Наступним кроком в ослабленні несуперечливості є звільнення від останньої вимоги. Це відповідає **несуперечливості FIFO** (*FIFO consistency*), яка відповідає такій умові: *операції запису, які здійснюються одиничним процесом, спостерігаються всіма іншими процесами в тому порядку, у якому вони здійснюються, але операції запису, що відбуваються в різних процесах, можуть спостерігатися різними процесами в різному порядку.*

Несуперечливість FIFO у випадку розподілених систем зі спільно використовуваною пам'яттю називають також несуперечливістю PRAM (PRAM consistency). Скорочення PRAM походить від Pipelined RAM (конвеєрна оперативна пам'ять), тому що запис, який виконується одним процесом, може бути конвеєризований через те, що процес не має втрачати швидкість, очікуючи закінчення однієї операції запису, щоб розпочати іншу. Наведена послідовність (рис. 1.9) подій допустима для сховища даних з несуперечливістю FIFO, але заборонена для кожної більш строгої моделі з тих, що розглядались.

Несуперечливість FIFO є простою у реалізації, але вона не гарантує дотримання порядку, у якому операції запису спостерігатимуться в різних процесах, крім порядку проходження двох або більше операцій запису, що належать одному процесу, тобто у цій моделі всі операції запису в усіх процесах є паралельними. Ця модель може реалізовуватися шляхом іменування кожної операції парою (процес,

черговий номер) і здійсненням операцій запису кожного із процесів у порядку їх номерів.

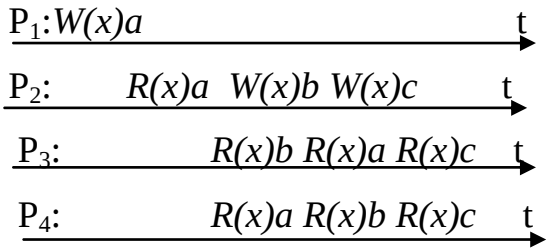


Рис. 1.9. Допустима послідовність подій для несуперечливості FIFO

1.2.5. Слабка несуперечливість

Хоч несуперечливість FIFO і дозволяє підвищити продуктивність у порівнянні з більш строгими моделями несуперечливості, у багатьох випадках вона залишається занадто строгою, оскільки вимагає, щоб операції запису одного процесу у всіх інших процесах спостерігалися в тому ж самому порядку. Разом з тим, не для всіх процесів потрібно спостерігати всі операції запису, а також їх порядок. Розглянемо випадок, коли процес усередині критичної області заносить запис в репліковану базу даних. Хоча інші процеси не працюють з новими записами до виходу цього процесу із критичної області, система керування базою даних не знає, перебуває процес у критичній області або ні, а тому може поширити зміни на всі копії бази даних.

Найкращим рішенням у цьому випадку було б дозволити процесу покинути критичну область і потім переконатися, що остаточні результати розіслані туди, куди потрібно, не звертаючи уваги на те, розіслані всім копіям проміжні результати або ні. Це можна зробити за рахунок введення **змінної синхронізації** (*synchronization variable*), яка має тільки одну асоційовану з нею операцію, *synchronize(S)*, що синхронізує всі локальні копії сховища даних. Нагадаємо, що процес *P* здійснює операції тільки з локальною копією сховища. У ході синхронізації

сховища даних всі локальні операції запису процесу *P* поширюються на інші копії, а операції запису інших процесів – на копію даних *P*.

Використовуючи змінні синхронізації для часткового визначення несуперечливості, приходимо до поняття **слабкої несуперечливості** (weak consistency). Модель слабкої несуперечливості має три властивості:

Доступ до змінних синхронізації, асоційованих зі сховищем даних, виконується на умовах послідовної несуперечливості.

Зі змінною синхронізації не може бути зроблена жодна операція до повного завершення всіх операцій запису, які їй передують, де б вони не виконувались.

З елементами даних не може бути зроблена жодна операція до повного завершення всіх операцій зі змінними синхронізації.

Вимога п.1 означає, що всі процеси можуть спостерігати за всіма операціями над змінними синхронізації, які з погляду цих процесів відбуваються в однаковому порядку. Інакше кажучи, якщо процес *P1* викликає операцію *synchronize(S1)* у той же самий час, коли процес *P2* викликає операцію *synchronize(S2)*, результат буде точно таким же, якщо б операція *synchronize(S1)* була викликана раніше операції *synchronize(S2),* або навпаки.

Вимога п. 2 вказує на те, що операція синхронізації очищає конвеєр, тобто прискорює виконання всіх операцій читання, які в цей момент відбуваються та є частково завершеними або завершеними в деяких локальних копіях, а також завершує їх в усіх інших процесах. Коли синхронізація закінчується, гарантовано закінчуються всі попередні їй операції запису. Виконуючи синхронізацію після зміни спільно використовуваних даних, процес переносить нові значення в усі локальні копії сховища.

Вимога п.3 означає, що при доступі до елементів даних (як для запису, так і для читання) всі попередні процеси синхронізації мають

бути завершеними. Виконавши перед операцією читання спільно використовуваних даних синхронізацію, процес може бути впевнений, що одержить актуальні на даний момент значення.

На відміну від попередніх моделей несуперечливості, слабка несуперечливість реалізує несуперечливість груп операцій, а не окремих операцій читання та запису. Ця модель найбільш широко використовується, якщо окремі процедури доступу до загальних даних відбуваються не часто, а більша частина операцій доступу поєднана в групи (багато операцій за короткий строк, а потім нічого протягом довгого часу).

Інша важлива відмінність від попередніх моделей несуперечливості полягає в тому, що обмеження визначаються тільки часом підтримки несуперечливості. У випадку слабкої несуперечливості дотримується послідовна несуперечливість між групами операцій, а не між окремими операціями. Для виділення цих груп використовуються змінні синхронізації.

1.2.6. Вільна несуперечливість

Слабка несуперечливість має таку проблему: коли здійснюється доступ до змінної синхронізації, сховище даних не знає, це відбувається тому, що процес закінчив запис спільно використовуваних даних, або, навпаки, почав читання даних. Відповідно, сховище може ініціювати дії, необхідні в обох випадках, наприклад, переконатися, що всі локально ініційовані операції запису завершено (тобто зміни поширені на всі копії) і враховано всі операції запису з інших копій. Для того, щоб сховище даних мало розрізняти операції входу у критичну область і виходу з неї, необхідно ввести або два типи змінних, або два типи операцій синхронізації, замість одного.

Вільна несуперечливість (*release consistency*) реалізує два типи операцій синхронізації. Операція *захвату (acquire)* використовується для повідомлення сховищу даних про вхід у критичну область, а операція *звільнення (release)* фіксує стан обчислювального процесу, за якого критична область покинута. Ці операції можуть бути реалізовані одним із двох способів: по-перше, звичайними операціями над спеціальними змінними; по-друге, спеціальними операціями. Для цього використовують вставку в програму відповідного додаткового коду, який реалізує, наприклад, виклик бібліотечних процедур *acquire* і *release* або процедур *enter_critical_region* і *leave_critical_region*.

У випадку вільної несуперечливості незалежно від критичних областей можна використати бар'єри. **Бар'єр** (*barrier*) – це механізм синхронізації, який випереджає будь-який процес на початку фази програми під номером *n+1* до того, як всі процеси закінчать фазу *n.* Коли процес підійде до бар'єра, він повинен дочекатися, поки до нього не «підтягнуться» і всі інші процеси. Коли останній із процесів підійде до бар'єра, всі спільно використовувані дані синхронізуються, і процеси продовжують свою роботу. Відправлення від бар'єра виконується по захвату, а прихід до бар'єра – по звільненню.

Додаткого до цих операцій синхронізації також можливі читання та запис спільно використовуваних даних. Операції захвату і звільнення не можуть застосовуватися до всіх даних сховища. Вони можуть охороняти тільки окремі спільно використовувані дані, у цьому випадку тільки ці дані залишаються несуперечливими. Спільно використовувані дані, що зберігають свою несуперечливість, називаються **захищеними** (*protected*).

Сховище даних з вільною несуперечливістю гарантує, що при захваті процесу сховище при необхідності актуалізує всі локальні копії захищених даних і вони стануть несуперечливими по відношенню до своїх віддалених копій. Коли відбудеться звільнення, змінені захищені

дані будуть поширені на інші локальні копії сховища. Захват не гарантує, що локальні зміни будуть негайно розіслані іншим локальним копіям. Відповідно, звільнення не обов'язково імпортує зміни з інших копій.

Щоб зрозуміти поняття вільної несуперечливості, розглянемо її можливу реалізацію для бази даних, яка реплікується. Виконуючи захват, процес посилає повідомлення центральному менеджерові синхронізації, яке запитує дозвіл на виконання захвату окремого блокування. У відсутності інших процесів, які запитують дозвіл на виконання захвату, запит задовольняється і захват відбувається. Після цього виконується довільна послідовність локальних операцій читання і запису. Жодна із цих операцій не поширюється на інші копії бази. У процесі звільнення модифіковані дані розсилаються іншим копіям, які використовують ці дані. Після того, як кожна копія підтвердить одержання цих даних, центральний менеджер синхронізації отримує повідомлення про звільнення, що відбулося. Таким чином, довільне число операцій читання і запису спільно використовуваних даних супроводжується фіксованими додатковими витратами. Захвати і звільнення при різних блокуваннях відбуваються незалежно один від одного.

Описаний централізований алгоритм вирішує проблему несуперечливості даних в їх розподілених копіях, але це не єдиний підхід. Розподілене сховище даних є вільно несуперечливим за умови виконання ним трьох правил:

− перед виконанням операцій читання або запису спільно використовуваних даних всі попередні захвати цього процесу мають бути повністю закінчені;

− перед виконанням звільнення всі попередні операції читання й запису даного процесу мають бути повністю закінчені;

− доступ до змінних, які синхронізуються, має використовувати несуперечливість FIFO (послідовна несуперечливість не потрібна).

Якщо всі правила виконані й процеси правильно (тобто попарно) використовують захвати і звільнення, то результат будь-якого виконання не буде відрізнятися від порядку, характерного для послідовно несуперечливих сховищ. У результаті, блокування операцій над спільно використовуваними даними буде атомарним завдяки примітивам захвата й звільнення, які не дапускатимуть чергування.

У вільної несуперечливості є така реалізація, як «**ледача» вільна несуперечливість** (*lazy release consistency*). При звичайній вільній несуперечливості, що далі називатиметься «**енергійною» вільною несуперечливістю** (*eager release consistency*), при звільненні процес, який виконує звільнення, розсилає всі модифіковані дані всім процесам, які вже мають копії цих даних, і тому потенційно можуть бути зацікавленими в їх оновленій версії. Не існує способу вказати, потрібна оновлена версія даних певному процесу або ні, і тому з точки зору надійності оновлені дані одержують всі такі процеси.

Розіслати всім процесам усі дані нескладно, але це неефективно. При «ледачій» вільній несуперечливості в момент звільнення дані нікуди не розсилаються. Замість цього в момент захвата процес, який намагається зробити захват, має одержати найбільш актуальні дані від процесу або процесів, у яких вони зберігаються. Для визначення того, що ці елементи даних дійсно були передані, використовується протокол оцінок часу.

У багатьох програмах критична область розташовується всередині циклу. У випадку «енергійної» вільної несуперечливості звільнення відбувається при кожному проході циклу, при цьому всі модифіковані дані розсилаються всім процесам, які підтримують їх копії. Цей алгоритм є критичним до пропускної здатності каналів і викликає неминучі затримки. У варіанті «ледачої» вільної несуперечливості в

момент звільнення ніякі дані не розсилаються. При наступному захваті процес визначає, що вже має всі необхідні дані, а тому йому не потрібно генерувати ніяких повідомлень. У результаті, при «ледачій» вільній несуперечливості, до того, як інший процес не зробить захват, мережевий трафік взагалі не генерується. Повторювані пари операцій захват-звільнення, що відбуваються в тому ж самому процесі за умови відсутності спроб доступу до даних ззовні, не викликають ніякого навантаження на мережу.

1.2.7. Поелементна несуперечливість

Ще одна модель несуперечливості, створена для використання в критичних областях, – **поелементна несуперечливість** (*entry consistency*). Як і у випадку вільної несуперечливості, вона потребує вставки коду для захвату й звільнення на початку і в кінці критичної області. Однак на відміну від вільної несуперечливості, поелементна несуперечливість додатково вимагає, щоб кожен окремий елемент спільно використовуваних даних був асоційований зі змінною синхронізації – блокуванням або бар'єром. Якщо необхідно, щоб до елементів масиву відбувався незалежний паралельний доступ, то різні елементи масиву мають бути асоційовані з різними блокуваннями. Коли відбувається захват змінної синхронізації, несуперечливими стають тільки ті дані, які асоційовані із цією змінною синхронізації. Поелементна несуперечливість відрізняється від «ледачої» вільної несуперечливості тим, що в ній відсутній зв'язок між спільно використовуваними елементами даних і блокуваннями або бар'єрами, тому у процесі захвату необхідні змінні визначаються емпірично.

Завдяки зв'язуванню списку спільно використовуваних елементів даних зі змінними синхронізації, знижуються накладні витрати на захват і звільнення змінних синхронізації тому, що ці операції

виконуються з декількома синхронізованими елементами даних. Це також дозволяє, збільшуючи ступінь паралелізму, мати менше критичних областей, до яких одночасно виконується доступ та які включають у себе групи спільно використовуваних елементів даних, що не перетинаються. Це потрубує додаткових зусиль під час керування процесом паралельного доступу й призводить до складності зв'язування всіх поділюваних елементів даних зі змінними синхронізації.

Змінні синхронізації використовуються наступним чином. Кожна змінна синхронізації має поточного власника – процес, який захопив її останнім. Власник може багаторазово входити в критичні області й виходити з них, не посилаючи в мережу ніяких повідомлень. Процес, який не є в цей час власником змінної синхронізації, але має захопити її, повинен послати поточному власникові повідомлення, щоб запитати право власності й поточні значення даних, асоційованих зі змінною синхронізації. Крім того, кілька процесів можуть одночасно володіти змінною синхронізації, але не в ексклюзивному режимі. Це означає, що вони можуть прочитати асоційовані зі змінною синхронізації дані, але не записати їх.

Формально сховище даних забезпечує поелементну несуперечливість, якщо воно задовольняє трьом умовам**:**

− захват процесом доступу до змінної синхронізації неможливий доти, доки не здійснені всі оновлення спільно використовуваних даних цього процесу;

− поки один із процесів має ексклюзивний доступ до змінної синхронізації, ніякий інший процес не може захопити цю змінну синхронізації, у тому числі й не ексклюзивно;

− після ексклюзивного доступу до змінної синхронізації не ексклюзивний доступ будь-якого іншого процесу до цієї змінної

синхронізації заборонений, поки це не буде дозволено власником цієї змінної.

Перша умова означає, що якщо процес робить захват, то захват не може бути виконаний (тобто не можна передати керування наступній інструкції) доти, доки всі контрольовані загальні дані не стануть несуперечливими, тобто при захваті мають бути візуалізовані всі зміни, які зроблено в цих даних віддаленими процесами.

Друга умова означає, що перед оновленням елемента спільно використовуваних даних процес має увійти в критичну область в ексклюзивному режимі, щоб гарантувати, що ніякий інший процес у той же самий час не змінює ці дані.

Третя умова означає, що якщо процес хоче увійти в критичну область в не ексклюзивному режимі, він має спочатку перевірити, що власник змінної синхронізації, який відслідковує цю критичну область, одержав актуальну копію даних.

Однією із проблем програмування для поелементної несуперечливості є правильне зв'язування даних зі змінними синхронізації. Один зі способів рішення цієї проблеми полягає у застосуванні розподілених спільно використовуваних об'єктів. Це виконується наступним чином: кожен розподілений об'єкт має асоційовану з ним змінну синхронізації, яка надається базовою розподіленою системою при створенні розподіленого об'єкта, але ця змінна повністю прихована від клієнта. Коли клієнт звертається до методу розподіленого об'єкта, базова система спочатку виконує захват асоційованої з об'єктом змінної синхронізації. У результаті найактуальніші значення стану об'єкта, які можна реплікувати і розподілити по декількох машинах, передаються клієнтській копії цього об'єкта. У цей момент відбувається звертання до об’єкта, оскільки об'єкт залишається заблокованим від паралельних операцій.

Після закінчення звертання потрібно виконати внутрішню операцію звільнення, яка деблокує об'єкт для подальших операцій.

У результаті всі звертання до розподіленого, спільно використовуваного об'єкта послідовно несуперечливі. Клієнтське програмне забезпечення не має обробляти змінні синхронізації, оскільки вони повністю підтримуються базовою розподіленою системою. У той же час кожен об'єкт виявляється автоматично захищеним від одночасного виконання паралельних запитів.

1.2.8. Порівняння моделей несуперечливості

В попередніх розділах описані основні моделі несуперечливості (моделі несуперечливості даних), далі розглядатимуться інші більш складні моделі. Вони відрізняються обмеженнями, складністю реалізації, простотою програмування й продуктивністю. Строга несуперечливість є найбільш обмеженою, але оскільки її реалізація в розподілених системах, по суті, неможлива, то вона ніколи в них не застосовується.

Лінеаризуємість – більш слабка модель несуперечливості, заснована на ідеї синхронізованих годин. У разі її застосування робити висновок про коректність виконання паралельних програм простіше, але все ж таки занадто складно для того, щоб її можна було використати для побудови розподіленого прикладного програмного забезпечення в індутрії. З цього погляду кращою моделлю є послідовна несуперечливість, яка широко використовується. Однак для неї характерною є проблема низької продуктивності. Єдиний спосіб підвищити показники продуктивності - це послабити модель несуперечливості. Деякі з можливостей послаблення впливу вимоги несуперечливості ілюструє табл. 1.1. Моделі несуперечливості перераховані в приблизному порядку зниження обмежень.

Таблиця 1.1. Моделі несуперечливості, які не потребують операцій синхронізації

Несуперечливість	**Опис**
Строга	Абсолютна впорядкованість у часі всіх звертань до спільно використовуваної пам'яті
Лінеаризуємість	Всі процеси спостерігають всі звертання до спільно використовуваної пам'яті в однаковому порядку. Запити впорядковані відповідно до (неунікальних) глобальних відміток часу
Послідовна	Всі процеси спостерігають всі запити до спільно використовуваної пам'яті, зв'язані причинно-наслідковим зв'язком, в однаковому порядку. Запити не впорядковані за часом
Причинна	Всі процеси спостерігають всі запити до спільно використовуваної пам'яті, зв'язані причинно-наслідковим зв'язком, в однаковому порядку
FIFO	Всі процеси спостерігають операції запису будь-якого процесу у відповідності з порядком їх виконання. Операції запису різних процесів можуть спостерігатися різними процесами в різному порядку

Причинна несуперечливість і несуперечливість FIFO є ослабленими моделями, у яких відсутній глобальний контроль за порядком виконання операцій. Різними процесами послідовність виконання операцій спостерігається як різна. Ці дві моделі несуперечливості відрізняються одна від другої тим, що по різному приймається рішення щодо допустимості послідовності виконання операцій.

Суть іншого підходу полягає у введенні явних змінних синхронізації, як у разі слабкої, вільної й поелементної несуперечливості. Характеристику цих трьох моделей наведено в табл. 1.2. Коли процес виконує операцію зі звичайним елементом спільно використовуваних даних, не визначено, коли зміни в даних, які ним зроблено, зможуть побачити інші процеси. Зміни поширюються тільки у разі використання явної синхронізації. Дані три моделі несуперечливості розрізняють за способом синхронізації, але у всіх випадках процес може виконувати множинні операції читання й запису

в критичній області без реального перенесення даних. Після закінчення обробки даних у критичній області результат передається іншим процесам або зберігається в кінцевому виді, очікуючи, доки інший процес не запитає ці дані.

Таблиця 1.2. Моделі несуперечливості, що використовують операції синхронізації

Несуперечливість	**Опис**
Слабка	Спільно використовувані дані вважають несуперечливими тільки після синхронізації
Вільна	Спільно використовувані дані вважають несуперечливими після виходу із критичної області
Поелементна	Спільно використовувані дані, які відносяться до даної критичної області, вважають несуперечливими при вході в цю область

Слабка, вільна та поелементна несуперечливості потребують використання додаткового програмного забезпечення, яке утворює ілюзію, що сховище даних має послідовну несуперечливість, але у такому разі коректність виконання усіх операцій доступу залежить від якості розробки цього програмного забезпечення. Дані три моделі, які використовують явну синхронізацію, можна використати для підвищення продуктивності, однак цілком імовірно, що для прикладного програмного забезпечення з різних предметних областей, які накладають специфічні обмеження, ефективність такого підходу буде різною.

1.3. Моделі несуперечливості, орієнтовані на клієнтські процеси

Моделі несуперечливості, описані в попередньому розділі, орієнтовані на створення несуперечливого представлення сховища даних. При цьому зробилено важливе припущення, що паралельні процеси одночасно змінюють дані в сховищі й необхідно зберегти несуперечливість сховища в умовах такої паралельності.

Приклад. Так у випадку поелементної несуперечливості з використанням об'єктів сховище даних гарантує, що при звертанні до об'єкта процес одержить копію об'єкта, яка відбиває всі зміни, що відбулися з ним, у тому числі й зроблені іншими процесами. Під час звертання гарантується також, що процесу не перешкодить ніякий інший об'єкт, тобто процесу, який звертається, буде надано доступ, захищений механізмом взаємного виключення.

Можливість здійснювати паралельні операції над спільно використовуваними даними в умовах послідовної несуперечливості є базовою для розподілених систем. Через невисоку продуктивність послідовна несуперечливість може гарантуватися тільки у випадку використання таких механізмів синхронізації як транзакції або блокування.

Розглянемо спеціальний клас розподілених сховищ даних, які характеризують відсутністю одночасних змін або легкістю їх впорядкованого проведення в тому випадку, якщо одночасні зміни все-таки трапляться. Такі сховища даних відповідають дуже слабкій моделі несуперечливості, яку називають потенційною несуперечливістю. Завдяки введенню спеціальних моделей несуперечливості, орієнтованих на клієнтські обчислювальні процеси, відносно просто приховується багато порушень несуперечливості.

1.3.1. Потенційна несуперечливість

Ступінь, у якій процеси дійсно працюють паралельно, і ступінь, у якій дійсно гарантуватиметься несуперечливість, можуть бути різними. Існує багато випадків, коли паралельність потрібна обмежено.

Приклад. У багатьох системах керування базами даних більшість процесів не виконує змін даних, використовуючи тільки операції читання. Зміну даних здійснє або один, або, у крайньому випадку, кілька процесів. Проблема полягає в тому, як швидко ці зміни можуть стати доступними процесам, які здійснюють тільки читання даних.

Як інший приклад розглянемо глобальну систему іменування DNS. Простір імен DNS розділений на домени, кожен домен має джерело іменування, що функціонує, як власник цього домену. Тільки це джерело може оновити свою частину простору імен. Конфлікт двох операцій, які хотіли б одночасно оновити ті ж самі дані (тобто конфлікт подвійного запису), неможливий. Єдина ситуація, яка може виникнути, – це конфлікт читання-запису. У такому разі, якщо конфлікт виникає, поширення змін можливо виконувати поступово, тобто процеси читання виявлятимуть зміни, які відбулися, тільки через певний час після того, як вони насправді відбулися.

Ще один приклад - World Wide Web. Фактично web-сторінки завжди змінюються однією людиною - web-майстром або дійсним власником сторінки. За такої умови конфлікти подвійного запису не виникають. З іншого боку, для підвищення ефективності функціонування браузери і проксі-сервери часто конфігурують так, щоб вони зберігали завантажені сторінки в локальному кеші і повертали при наступному запиті саме їх. Важлива особливість обох типів систем web-кеширування полягає у тому, що вони можуть повертати застарілі web-сторінки. Кешовані сторінки, які зберігаються на web-сервері та повертаються у відповідь на запит клієнта, можуть не відповідати їх новим версіям, але навіть у такому разі багато користувачів вважають таку невідповідність прийнятною.

Ці приклади можуть розглядатися як випадки розподілених і реплікованих баз даних (великих), нечутливих до відносно високого ступеня порушення несуперечливості. Зазвичай. у них тривалий час не відбуваються зміни даних і всі репліки поступово стають несуперечливими. Таку форму несуперечливості називають **потенційною несуперечливістю** (*eventual consistency*).

Потенційно несуперечливі сховища даних мають властивість: *у разі відсутності змін всі репліки поступово стають ідентичними.* Потенційна несуперечливість, по суті, має гарантувати, що зміни відбуватимуться в усіх репліках. Конфлікти подвійного запису часто відносно легко вирішуються за припущення, що вносити зміни може лише невелика група процесів, тому реалізація потенційної несуперечливості не вимагає значних витрат.

Потенційно несуперечливі сховища даних добре працюють, якщо клієнт здійснює доступ до однієї репліки. Однак, при доступі до різних реплік виникають проблеми, які проілюстровано схемою доступу мобільного користувача до розподіленої бази даних, представленої на рис. 1.10.

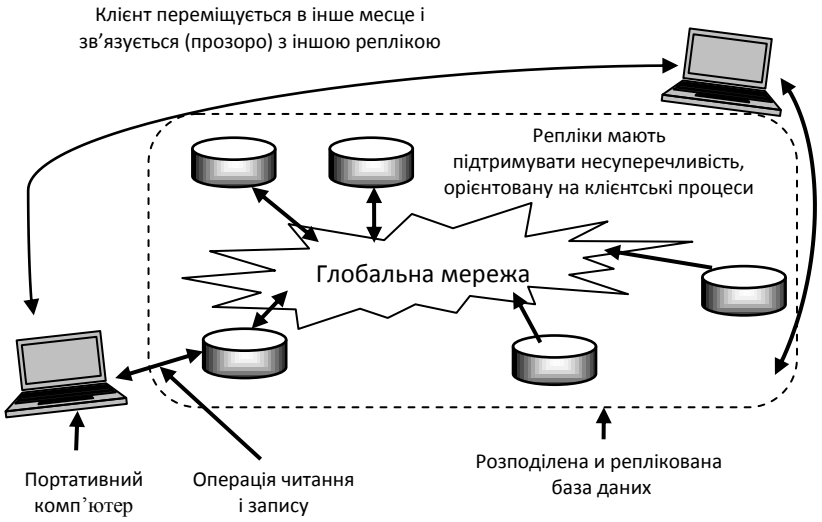


Рис. 1.10. Принцип доступу мобільного користувача до різних реплік розподіленої бази даних

Приклад. Мобільний користувач працює з базою даних, прозоро приєднуючись до однієї з її реплік. Програма, яка функціонує на портативному комп'ютері користувача, не знає, з якою саме реплікою вона працює. Припустімо, користувач зробив кілька операцій зміни даних та відключився від бази. Пізніше він знову почав працювати з базою даних, можливо, після переміщення в інше місце або з іншого пристрою доступу. У цей момент користувач може приєднатися до іншої репліки, як показано на рис.1.10. Однак, якщо зміни, зроблені в базі даних раніше, ще не поширені на цю репліку, то користувач виявить протиріччя даних у базі даних. Зокрема, він очікуватиме, що зроблені раніше зміни вже внесені, а фактично виявить, що нічого не змінилося. Цей приклад типовий для потенційно несуперечливих сховищ даних, де проблемою є те, що користувач може іноді працювати з різними репліками.

Гострота проблеми можливого доступу до різних реплік даних знижується шляхом введення **несуперечливості, орієнтованої на клієнтські процеси** (*client-centric consistency*), яка надає одному клієнту гарантії несуперечливості при доступі до сховища даних,

стосовно ж паралельного доступу інших клієнтів ніяких гарантій не надається. Моделям несуперечливості, орієнтованим на клієнтські процеси, дала початок робота над Bayou – системою керування базами даних, розробленої для мобільних обчислень. У цій системі передбачається, що мережні підключення ненадійні й мають різні проблеми із продуктивністю. До цієї категорії підключень відносять бездротові мережі й глобальні мережі, такі як Internet.

В системі Bayou розрізняють чотири основних моделі несуперечливості, які реалізуються на базі сховища даних, розподіленого декількома машинами. Коли процес одержує доступ до сховища даних, він зазвичай зв'язується з локальною (або найближчою) копією даних, це може бути будь-яка копія. Всі операції читання й запису здійснюються з локальною копією, зміни поступово поширюються й на інші копії. Для спрощення приймемо, що елементи даних мають асоційованого з ними власника, яким є єдиний процес, який має право змінювати їх. Це спрощення дозволяє усунути конфлікти подвійного читання. Моделі несуперечливості, орієнтовані на клієнтські процеси, описують за допомогою такої нотації. Нехай $x_i[t]$ означає версію елемента даних x на локальній копії L_i у момент часу t . Версія $x_i[t]$ – результат серії операцій запису, зроблених в L_i після їх ініціалізації. Позначимо цю серію операцій як $WS(x_i[t])$, а операції із серії $WS(x_i[t_1])$, які було зроблено в локальній копії L_j пізніше у час t_2, запишемо як $WS(X_i[t_1];X_j[t_2])$. У разі, якщо тимчасова черговість операцій буде зрозуміла з контексту, індекс символу часу опускається.

1.3.2. Несуперечливість монотонного читання

Перша з моделей несуперечливості, орієнтована на клієнтські процеси, – **монотонне читання**. Сховище даних забезпечує несуперечливість монотонного читання (*monotonic-read consistency*), у

разі виконання такої умови: якщо процес читає значення елемента даних *x*, то будь-яка наступна операція читання *x* завжди повертає те ж саме або подальше нове значення, тобто, несуперечливість монотонного читання гарантує, що якщо процес у момент часу *t* бачить певне значення *x,* то пізніше він ніколи не побачить старе значення *x.*

Приклад. Розглянемо розподілену базу даних електронної пошти. У цій базі даних поштова скринька кожного користувача може бути розподіленою декількома машинами і реплікована. Кореспонденція додається в поштові скриньки всіх реплік. Однак зміни поширюються повільно (за запитом). Дані пересилаються копії бази тільки в тому випадку, коли ці дані потрібні цій копії для підтримки несуперечливості. Припустімо, що користувач читає свою пошту в Харкові. Також припустімо, що читання пошти не впливає на поштову скриньку, тобто повідомлення не видаляються, не зберігаються у вкладених каталогах, не позначаються як прочитані й т.п. Коли користувач після цього летить до Києва і знову відкриває свою поштову скриньку, то несуперечливість монотонного читання гарантує, що повідомлення, які він бачив у своїй поштовій скриньці в Харкові, він побачить й у Києві.

1.3.3. Несуперечливість монотонного запису

У багатьох випадках важливо, щоб по всіх копіях сховища даних у правильному порядку поширювалися операції запису. Це здійснюють, використовуючи **несуперечливість монотонного запису** (*monotonic-write consistency*). Сховище даних забезпечує несуперечливість монотонного запису, якщо виконується така умова**:** операція запису процесу в елемент даних *x* завершується раніше ніж кожна з наступних операцій запису цього процесу в елемент *x*.

Завершення операції запису означає, що копія, над якою виконується наступна операція, відображає результат попередньої операції запису, зробленої тим же процесом, і у такому разі не має значення, де ця операція була ініційована, тобто, операція запису в копію елемента даних *x* виконується тільки у тому випадку, якщо ця

копія відповідає результатам попередньої операції запису, яку виконано над іншими копіями *x*.

Відзначимо, що несуперечливість монотонного запису схожа з однією з моделей несуперечливості, орієнтованою на дані, – несуперечливісттю FIFO. Сутність несуперечливості FIFO полягає в тому, що операції запису одного процесу завжди виконуються в правильній черговості. Таке обмеження на черговість застосовують й у разі монотонного запису, за винятком того, що у разі монотонного запису розглядають один процес, а не набір паралельних процесів.

Актуалізація копії *x* не є обов'язковою, якщо кожна операція запису повністю змінює існуюче значення *x*. Однак операції запису часто виконуються тільки над частиною елемента даних.

Приклад. Розглянемо бібліотеку підпрограм. У багатьох випадках оновлення бібліотеки полягає в заміні однієї або декількох функцій, а результатом цього є нова версія бібліотеки. За умови несуперечливості монотонного запису надають гарантії того, що якщо зміни відбуваються в одній з копій бібліотеки, то спочатку будуть внесені всі зміни, що відбулися раніше, та не внесені ще в бібліотеку. Отже, бібліотека буде реально містити всі оновлення, що були в попередній версії цієї бібліотеки.

Можна відзначити, що, за визначенням несуперечливості монотонного запису, операції запису одного процесу виконують в тому ж порядку, у якому вони були ініційовані. В ослабленій формі несуперечливості монотонного читання результат операцій читання спостерігають тільки у тому разі, якщо всі попередні операції читання вже завершилися, але не обов'язково в тому порядку, у якому вони починалися. Таку несуперечливість застосовують тоді, коли операції запису комутативні, тобто їх впорядкованість не є необхідною.

1.3.4. Несуперечливість читання власних записів

Існує ще одна модель несуперечливості, орієнтована на клієнтські процеси, яка є досить подібною до несуперечливості монотонного запису. Сховище даних має властивість несуперечливості **читання власних записів** (*read-your-writes consistency*), якщо воно задовольняє такій умові: результат операцій запису процесу в елемент даних *x* завжди видимий для наступних операцій читання *x* цього ж процесу, тобто, операція запису завжди завершується раніше наступної операції читання одного й того ж процесу, де б не відбувалася ця операція читання.

Приклад. Відсутність несуперечливості читання власних записів часто виявляється під час відновлення web-сторінок з наступним переглядом результатів. Операції оновлення відбуваються за допомогою стандартного або будь-якого текстового редактора, який записує нову версію в спільно використовувану файлову систему web-сервера. Web-браузер користувача працює з тим же самим файлом, запитуючи його з локального web-сервера. Однак, після першого ж завантаження файлу сервер або браузер кешують його локальну копію для наступного доступу. Відповідно, якщо браузер або сервер повертає кешовану копію, а не оригінальний файл, то при відновленні web-сторінки користувач не в змозі побачити його результат.

Несуперечливість читання власних записів може гарантувати, що якщо редактор і браузер інтегровані в єдиний програмний комплекс, то при відновленні сторінки кеш оголошується неактуальним і у такому разі завантажується та відображається нова версія файлу.

Приклад. Те ж саме відбувається і у разі зміни паролів. Так, наприклад, для входу в цифрову бібліотеку в Web-мережі часто необхідно мати обліковий запис із відповідним паролем доступу до неї. Однак, під час зміни пароля для того, щоб запрацював новий пароль, може знадобитися кілька хвилин. У такому разі ці кілька хвилин бібліотека буде недоступною для користувачів. Затримка може виникнути через те, що для роботи з паролями використовується окремий сервер і наступне розсилання пароля (зашифрованого) серверам, на яких знаходиться бібліотека, може потребувати часу. Реалізація несуперечливості читання власних записів вирішує цю проблему.

1.3.5. Несуперечливість запису після читання

Остання модель несуперечливості, орієнтована на клієнтські процеси, – це модель, у якій зміни даних поширюють як результати попередньої операції читання. У такому разі сховище даних забезпечує **несуперечливість запису після читання** (*writes-follow-reads consistency*), якщо задовільняється така умова: операція запису в елемент даних *x*, яка виконується після операції читання *x* одного й того ж процесу, гарантує, що вона виконуватиметься над тим же самим або актуальнішим значенням *x*, яке було прочитано попередньою операцією, тобто будь-яка наступна операція запису в елемент даних *x,* яка виконується процесом, буде здійснюватися з копією *x,* що має останнє зчитане тим же процесом значення *x.*

Приклад. Несуперечливість запису після читання дозволяє гарантувати, що користувачі мережної групи новин побачать листи з відповідями на певний лист тільки пізніше отримання оригінального листа. Користувач спочатку читає лист *A*, потім він реагує на нього, посилаючи лист *B.* Відповідно до вимоги несуперечливості запису після читання, лист *B* буде розіслано в усі копії групи новин тільки після того, як туди буде послано лист *A.* Потрібно відзначити, що користувачі, які тільки читають листи, не мають потреби в особливій моделі несуперечливості, яка орієнтована на клієнтські процеси. Несуперечливість запису після читання забезпечить збереження в локальній копії відповіді на лист тільки у тому разі, якщо там уже є в наявності оригінал.

1.3.6. Реалізація

Реалізація несуперечливості, орієнтованої на клієнтстські процеси, відносно проста, якщо не брати до уваги питання продуктивності.

У примітивній реалізації несуперечливості, орієнтованій на клієнтстські процеси, кожній операції запису призначається сервером, який першим виконує цю операцію, глобально унікальний

ідентифікатор. У такому разі генерація глобально унікальних ідентифікаторів може реалізовуватися у вигляді локальної операції. Отже, для кожного клієнта відслідковують два набори ідентифікаторів запису:

набір читання для клієнта складається з ідентифікаторів запису, що відповідають операціям читання, які виконано клієнтом;

набір запису складається з ідентифікаторів операцій запису, які виконано клієнтом.

Несуперечливість монотонного читання реалізується таким чином. Коли клієнт здійснює операцію читання із сервера, цей сервер перевіряє набір читання клієнта на локальну присутність результатів всіх його операцій запису (розмір цього набору може викликати проблеми продуктивності). Якщо результат перевірки негативний, то він зв'язується з другими серверами, щоб актуалізувати дані до проведення операції читання. Операції читання також можуть бути передані серверу, на якому виконувалися операції запису. Після виконання операції читання подальші операції запису, проведені на обраному сервері й пов'язані з операціями читання, додаються до набору читання клієнта.

Точно визначається лише місце, де відбуваються операції запису, зазначені в наборі читання. Так, наприклад, ідентифікатор запису може містити в собі ідентифікатор сервера, який ініціював дану операцію, визначати сервер потрібно для фіксації операції запису в журналі, щоб її можна було повторити на іншому сервері. Крім того, операції запису мають здійснюватися в тому ж порядку, у якому вони були ініційовані. Впорядкованість уведена для того, щоб клієнт міг генерувати глобально унікальний послідовний номер, який включатиметься в ідентифікатор запису, такий, наприклад, як відмітка часу Лампорта. Якщо кожен елемент даних модифікується тільки його власником, то останній і зможе підтримувати послідовну нумерацію.

Несуперечливість монотонного запису реалізується аналогічно монотонному читанню. Щоразу під час ініціювання клієнтом нової операції запису на сервері цей сервер переглядає набір записів клієнта. Сервер підтверджує, що зазначені операції запису виконано першими й у правильному порядку. Після виконання наступної операції ідентифікатор запису цієї операції додається до набору записів. Актуалізація набору записів поточного сервера за допомогою набору записів клієнта може істотно збільшити час відгуку сервера.

Несуперечливість читання власних записів також вимагає, щоб сервер, на якому виконуються операції читання, мав доступ до всіх операцій запису з набору запису клієнта. Операції запису можна отримувати з інших серверів перед виконанням операції читання, навіть якщо це знижує час відгуку. Крім того, клієнтське програмне забезпечення може самостійно знайти сервер, на якому операції запису вже виконано й зазначено в наборі запису клієнта.

Виходячи з вищевикладеного, несуперечливість запису після читання можна реалізувати таким чином: спочатку обраний сервер актуалізується за допомогою операцій запису, які входять у набір операцій читання клієнта, а потім у набір запису додаються ідентифікатор операції запису й ідентифікатори з набору операцій читання, що дозволяє врахувати тільки но виконану операцію запису.

1.4. Протоколи розподілу

Розглянемо способи поширення оновлень, тобто розподілу оновлень репліками незалежно від підтримуваної ними моделі несуперечливості.

Реплікація (*replication*) – механізм синхронізації вмісту декількох копій об’єкту, наприклад бази даних, процес, під яким розуміють копіювання даних з одного джерела до другого (або до багатьох інших

джерел), та в зворотньому порядку. Під час реплікації зміни, які зроблено в будь-якій копії об’єкта, можливо розповсюдити в інші копії. Реплікація потрібна, щоб організувати роботу одного або декількох користувачів з одним і тим же документом, базою даних або іншими файлами на різних комп'ютерах незалежно, без одночасного доступу до файлів у разі, коли потрібно підтримувати деяку загальну версію змінюваних файлів, яка містить в собі всі зроблені незалежно останні виправлення. Іншими словами, реплікація – це процес створення копій файлів, між якими може здійснюватися обмін оновлюваними даними або об'єктами. Такі копії називаються **репліками**, а такий обмін - **синхронізацією.**

Приклад. Існує декілька засобів реплікації баз даних, проектів баз даних та інших файлів. Розглянемо приклад засобу Access, де використовуються такі засоби реплікації:

- портфельна реплікація – засіб операційної системи Microsoft Windows;
- реплікація баз даних та проектів засобами Access – вбудовані засоби Microsoft Access;
- реплікація за допомогою диспетчера реплікації Microsoft – повнофункціональне керування репліками, планування синхронізації і перегляду елементів набору реплік;
- реплікація файлів на сервері Web – засіб сервера Web фірми Microsoft, який дозволяє працювати з файлами, збереженими на вузлі Web, в автономному режимі (без підключення до сервера);
- програмна реплікація за допомогою інтерфейсів DAO і JRO.

Реплікація включає наступні дії:

- вибір засобу реплікації;
- створення реплік;
- синхронізація реплік;
- керування репліками.

Якщо розглядати реплікацію на прикладі СКБД Access, то реплікою в даному випадку називається кожна копія реплікованих баз даних. Кожна репліка бази даних містить загальний (для всіх реплік бази даних) набір таблиць, запитів, форм, звітів, сторінок доступу до даних, макросів і модулів. Зміни даних таблиці,

зроблені в одній з реплік, передаються в інші репліки. Кожна репліка може також містити локальні об'єкти, які існують тільки в цій репліці.

Окрема репліка є компонентом набору реплік і допускає синхронізацію з іншими репліками в наборі. У наборі реплік виділяється **головна репліка.**

Синхронізація реплік – процес оновлення двох компонентів набору реплік, за яким відбувається взаємний обмін оновленими записами та об'єктами. Після синхронізації двох компонентів набору реплік зміни з кожної репліки відображаються в іншій репліці.

Часткова репліка – база даних, що містить підмножину записів повної репліки. За допомогою часткової репліки користувач має можливість встановлювати фільтри і задавати відносини, що визначають, яка підмножина записів повної репліки має входити до бази даних.

Область видимості. Репліки у відповідністю за областю видимості поділяють на три типи: глобальні, локальні і анонімні (табл.1.3).

Таблиця 1.3. Типи реплік

Тип репліки	Опис
Глобальна (*global*) репліка	Репліка, у відповідності з якою створюватимуться репліки всіх інших типів. При реплікації бази даних перша створювана репліка (основна реплікою) є глобальною реплікою. Зміни, внесені в глобальну репліку, відслідковуються повністю; можливий обмін цими змінами з будь-якою іншою глобальною реплікою в наборі. Глобальна репліка може також обмінюватися змінами з будь-якою локальною або анонімною реплікою. Розгалуджувачем називається глобальна репліка, з якою всі репліки в наборі синхронізують свої зміни
Локальна (*local*) репліка	Репліка, в якій фіксуються тільки дані з цієї репліки і не відображаються дані з інших реплік в наборі, в тому числі з основної репліки. Локальна репліка синхронізує свої дані з розгалужувачем, яким є глобальна репліка. Синхронізація локальних реплік з іншими репліками в наборі не дозволена. Якщо зміни в локальній репліці конфліктують з глобальною

	реплікою-розгалужувачем, ці зміни автоматично втрачаються в будь-якому процесі усунення конфліктів
Анонімна (*anonymous*) репліка	Особливий тип реплік в базах даних, для яких не ведеться фіксація окремих користувачів. Анонімні репліки особливо зручні при роботі в Інтернеті, коли очікується взаємодія реплік багатьма користувачами

Пріоритет репліки – це спеціальна характеристика репліки бази даних, яка визначає відносний пріоритет репліки в наборі реплік. Пріоритет позначають цілим позитивним числом. Пріоритет реплік враховується при вирішенні конфліктів, які виникли в процесі синхронізації реплік.

Приклад. Розглянемо процес створення реплік в базі даних Access. Головна і локальна репліки бази даних створюються в Access однаково, за допомогою команд меню «Сервіс», «Реплікація», «Створити репліку». Порядок створення:

1. Необхідно вибрати команду «Сервіс» → «Реплікація» → «Створити репліку» (*«Tools» → «Replication» → «Create Replica»*) відповідно.
2. З'явиться діалогове вікно з вимогою підтвердження закриття бази даних. Потрібно натиснути кнопку «Так (*Yes*)».
3. Далі з'явиться діалогове вікно «Розміщення нової репліки» (*«Location of New Replica»*), де можна вказати папку для розміщення нової репліки. Якщо потрібно, змінюється ім'я файлу репліки, яке задане за замовчуванням.
4. Щоб задати пріоритет репліки, необхідно натиснути кнопку «Пріоритет» *(«Priority»*), з'явиться діалогове вікно, де вводиться пріоритет репліки, який потрібно вказати, а потім натиснути кнопку «ОК».
5. Якщо потрібно заборонити користувачам видаляти записи з таблиць в репліці, встановлюється прапорець «Заборонити видалення» (*«Prevent delets»*).
6. Щоб задати область видимості репліки, необхідно вибрати потрібний елемент у списку, який розкривається командою меню «Тип файлу» (*«Save as type»*).
7. Далі необхідно натиснути кнопку «ОК». Почнеться процес створення репліки. Після його завершення може з'явитися пояснювальне повідомлення з текстом «Зміни структури бази даних допускаються тільки в головній репліці; зміни даних можуть виконуватися як в головній, так і в будь-якій іншій репліці набору».

Синхронізація реплік. Пряму синхронізацію використовують у випадках, коли репліки безпосередньо підключені до локальної мережі і розташовані у загальних мережних папках. Пряму синхронізацію не рекомендують використовувати у разі віддаленої синхронізації за допомогою сервера віддаленого доступу (RAS) або з'єднання віддаленого доступу.

Непряму синхронізацію доцільно застосовувати у разі роботи в автономному середовищі, наприклад на портативному комп'ютері.

Синхронізація через Інтернет є зручним способом синхронізації реплік у тих автономних середовищах, які мають доступ до Internet.

1.4.1. Розміщення реплік

Основна проблема проектування розподілених сховищ даних, яка має вирішуватись, - це визначення питання: коли, де і кому розміщати копії сховища. Розрізняють три різних типи копій: постійні репліки, репліки, ініційовані сервером, і репліки, ініційовані клієнтом.

Постійні репліки розглядають як вихідний набір реплік, які утворять розподілене сховище даних. У багатьох випадках число постійних реплік невелике.

Приклад. Одним з характерних прикладів постійної реплікації є web-сайт. Реплікація файлів web-сайтів в територіально-розподіленому середовищі зазвичай відбувається в одному із двох варіантів. У першому варіанті файли, в яких розміщено контент сайту, реплікуються на обмеженому числі серверів однієї локальної мережі. Коли надходить запит, він передається одному з серверів, де він і обробляється.

Другий тип реплікації файлів web-сайтів – це **створення дзеркал** (*mirroring*). У такому разі web-сайт копіюється на обмежену кількість розкиданих по всьому Інтернету серверів, які називають **дзеркальними** сайтами (*mirror sites*), або просто дзеркалами. У більшості випадків клієнти вибирають одне із дзеркал із запропонованого їм списку. Дзеркальні web-сайти зазвичай використовують

технологію кластерних web-сайтів, яка підтримуєь одну або кілька реплік з можливістю їх статичного конфігурування.

Подібну ж статичну організацію застосовують й для створення розподілених баз даних. Бази даних можуть бути репліковані й розподілені по декількох серверах, які разом утворюють кластер робочих станцій (*Cluster Of Workstations, COW*). Про такі кластери часто говорять як про **архітектуру без поділу** (*shared-nothing architecture*), підкреслюючи, що ні диски, ні оперативна пам'ять не використовуються процесорами спільно.

Разом з тим, бази даних можуть розподілятися й реплікуватися множиною географічно розкиданих місць. Така архітектура нерідко застосовується при побудові федеральних баз даних.

На противагу постійним реплікам, **репліки, ініційовані сервером**, є копіями сховища даних, які необхідні для підвищення продуктивності, їх створення ініціюється саме сховищем даних (його власником).

Приклад. Розглянемо web-сервер, який перебуває у Києві. Зазвичай такий сервер здатний досить швидко обробляти вхідні запити, але може трапитися так, що протягом декількох днів з невідомого віддаленого від сервера місця надходиметь значний потік запитів. (Такий потік у середовищі Web може бути викликаний багатьма причинами.) У такому разі доцільно динамічно створити в регіонах України, з якими працює сервер, кілька тимчасових реплік, що працюватимуть із вхідними запитами свого регіону. Такі репліки називають **кешами, що просуваються** (*push caches*).

Проблему динамічного розміщення реплік застосовано у службах web-хостингу. Ці служби, які, по суті, пропонують набір серверів (більше менш статичний), розкиданих по всьому Інтернету, підтримують і надають доступ до web-файлів, які належать третім особам. Для покращення якості послуг служби хостингу можуть динамічно реплікувати файли на сервери, для яких така реплікація збільшить продуктивність, тобто ближче до клієнтів або груп клієнтів.

Якщо уявити собі шлях руху даних від прикладного програмного забезпечення до дискової системи зберігання, то можна побачити три ключових місця, де можливі перехоплення даних і їх перенапрямок для подальшого запису на диски: хост (комп'ютер, на якому виконується прикладна програма), мережа зберігання даних (*SAN*) і контролер

дискового масиву. Виходячи з цього виділяють три відповідні типи реплікації: на рівні хосту, мережі зберігання даних та дискового масиву.

Реплікація на рівні хосту. У цьому разі завдання реплікації здійснює комп'ютер, на якому виконується прикладна програма з обробки даних. Цей спосіб є найбільш гнучким і, в той же час, найбільш складним у керуванні (особливо для великої групи серверів), він залежить від операційної системи й використовує обчислювальні ресурси прикладних програм серверів. Реплікацію на рівні хоста можна подати у вигляді спеціальної системи, прозорої для прикладного програмного забезпечення та операційної системи, яка дублює та перенапрямлює операції вводу/виводу, які виконуються на чотирьох відповідних підрівнях організації даних: специфічних для прикладної програми, блокового, файлового і логічних томів.

Підрівень прикладного програмного забезпечення. Специфічним способом реплікації на рівні хоста є реплікація на підрівні прикладного програмного забезпечення. Оскільки реалізація розподілених систем клієнт-сервер передбачає застосування реплікації даних, то після появи перших клієнт-серверних прикладних програм було розроблено й перші продукти для реплікації на підрівні прикладного програмного забезпечення. Таку технологію вперше використали (1984 р.) у проекті Lotus Notes. Технології реплікації на рівні прикладного програмного забезпечення першими застосували в корпорації IBM для БД IMS (1987 р.).

Головні недоліки такого способу реплікації полягають у його прив’язці до певного прикладного програмного забезпечення, необхідності купівлі ліцензій для резервних серверів і виконання процедур реплікації з використанням ресурсів сервера прикладного програмного забезпечення.

Блоковий підрівень. Найбільш гнучкою з погляду інтеграції в наявну інфраструктуру є реплікація на блоковому підрівні, яка не залежить від використовуваних файлових систем і способів зберігання даних і може застосовуватися як для локальної, так і для вилученої реплікацій. На жаль, кожен конкретний продукт такого підрівня реплікації зазвичай «прив'язаний» до певної операційної системи.

Сфера застосування: швидке відновлення після надзвичайних подій, створення «миттєвих знімків» і міграція даних. Приклади продуктів для різних ОС (Windows, AIX, Solaris, HP-UX, z/OS): Legato RepliStor, Veritas Storage Replicator, NSI Double-Take, IBM HAGEO й GeoRM, Sun StorEdge Availability Suite, Softek Replicator й TDMF.

Файловий підрівень. Реплікація на цьому підрівні використовується там, де необхідно синхронізувати невелику кількість файлів. Вона дозволяє робити вибіркову реплікацію даних та їх перенесення. Цей спосіб працює на рівні файлової системи, чим визначаються всі переваги й недоліки вказаного типу. Не придатний він для створення копії БД або заблокованих системних файлів, але, в той же час, реплікація на файловому підрівні підходить для прикладних програм, для яких необхідно забезпечити одночасний доступ до всіх копій файлів даних. Приклади продуктів: HP OpenView Storage Mirroring і протокол SUP (Software Update Protocol), розроблений в інституті Карнегі Меллона.

Підрівень логічних томів. Створення «дзеркальної» копії даних за допомогою менеджера логічних томів застосовується частіше для підвищення надійності доступу до даних на локальному рівні, ніж для створення вилучених копій або «миттєвих знімків». Такий спосіб також використовується для міграції даних, але не є кращим для цього виду робіт через певні архітектурні обмеження. Наприклад, якщо джерело даних розміщене на томі із чергуванням (*striped volume*), то, звичайно, й одержувач повинен бути на тому ж томі. Цей спосіб реплікації

широко застосовують завдяки його зручності, оскільки менеджера логічних томів зазвичай уже вбудовано в операційну систему. Приклади продуктів: Veritas Volume Manager, IBM LVM.

Одна з проблем реплік, створення яких ініційовано сервером, полягає в необхідності ухвалення рішення про те, де й коли будуть створюватися і знищуватися ці репліки. Алгоритм розроблено для підтримки web-сторінок, тому передбачено рідкі, порівняно із запитами на читання, оновлення. Як модулі даних використовуються файли.

Ще одним важливим типом є репліки, створювані з ініціативи клієнтів. Ініційовані клієнтом репліки часто називають **клієнтськими кешами** (*client caches*) або просто кешами (*caches*). По суті, кеш – це локальний пристрій зберігання даних, використовуваний клієнтом для тимчасового зберігання копії запитаних даних, при цьому відповідальність за керування кешем повністю покладається на клієнта. Сховище, з якого вилучаються дані, не робить нічого для підтримки несуперечливості кешованих даних. Однак існує багато випадків, у яких клієнт покладається на те, що сховище даних сповістить його про старіння кешованих даних. Клієнтський кеш використовується тільки для зменьшення часу доступу до даних. Звичайно, коли клієнт хоче одержати доступ до деяких даних, він зв'язується з найближчим сховищем з їх копією й зчитує звідти дані, які йому потрібні, або зберігає туди дані, які він щойно змінив. Продуктивність більшості операцій, що містять лише операції читання даних, можна підвищити, зберігаючи запитані клієнтом дані у близько розміщеному кеші, зокрема на машині клієнта або на окремій машині в тій самій локальній мережі, що й машина клієнта. Коли клієнт наступного разу звернеться до тих самих даних, він одержить їх із локального кешу. Ця схема відмінно працює, якщо у проміжку між запитами дані не змінювалися.

Час зберігання даних у кеші зазвичай обмежується, наприклад для того, щоб запобігти фатальному старінню інформації або просто

звільнити місце для нових даних. Якщо запитувані дані можуть бути отримані з локального кеша, то відбувається **кеш-попадання** (*cache hit*). Щоб підвищити кількість кеш-попадань, кеш можуть спільно використовувати декілька клієнтів, передбачаючи що дані, запитані клієнтом $З_1$, можуть знадобитися також й іншому поруч із ним клієнтові $З_2$. Те, наскільки це припущення правильно, значною мірою залежить від типу сховища даних.

Так у традиційних файлових системах файли взагалі рідко використаються спільно, тому створення загального кешу не потрібно. Однак загальний кеш web-сторінок виявляється дуже корисним, хоча підвищення продуктивності поступово припиняється через те, що сайтів, спільно використовуваних різними клієнтами, стає все менше порівняно із загальною кількістю web-сторінок.

У деяких випадках системні адміністратори застосовують додаткові рівні кешування, вводиться кеш, який спільно використовує велика кількість відділів або фірм, або навіть створюється загальний кеш для цілих регіонів, провінцій або країн.

Ще один підхід до кешування - помістити сервери (кеші) у певних точках глобальної мережі й дозволити клієнтові знайти найближчий. Коли сервер знайдений, то клієнт надсилає йому запит на збереження даних, які він звідкись одержав.

1.4.2. Способи й типи реплікації даних

Розглянемо два способи реплікації: **синхронний** й **асинхронний.**

У випадку синхронної реплікації, якщо певна репліка оновлюється, також має бути оновлено всі другі репліки того ж самого фрагменту даних в тій же самій транзакції. Операція запису деякої порції даних (репліки) вважається завершеною тільки після її підтвердження обома системами (основною та резервною). Абстрактно

це означає, що існує лише одна версія даних. Синхронний спосіб традиційно використовується там, де припустимий час простою вимірюється у хвилинах, або потрібним є негайне перемикання на резервну систему у випадку збою. Зазвичай його застосовують для невеликих відстаней, щоб зменшити негативний вплив на продуктивність і доступність системи.

У більшості систем синхронну реплікацію реалізують за допомогою тригерних процедур (можливо, прихованих та керованих системою). Синхронна реплікація має недолік через появу додаткового навантаження під час виконання всіх транзакцій, в яких оновлюються репліки. Крім того, вірогідним є виникнення проблем, пов’язаних з доступністю даних.

У разі асинхронної реплікації оновленння однієї репліки розповсюджують на другі через деякий час, а не у тій самій транзакції. Таким чином, під час виконання асинхронної реплікації виникає затримка, час очікування, на протязі якого окремі репліки можуть бути фактично неідентичними. Асинхронна реплікація дозволяє здійснювати запис до основної системи зберігання даних не чекаючи підтвердження виконання запису до резервної системи. Зазвичай такий тип реплікації використовують при розміщенні обох систем на значному віддаленні одна від другої.

У більшості систем асинхронна реплікація реалізується за допомогою читання журналу транзакцій або постійної черги тих оновлень, які підлягають розповсюдженню. Перевага асинхронної реплікації полягає у тому, що додаткові витрати реплікації не пов’язані з транзакціями оновлень, які можуть мати важливе значення для функціонування всієї системи та висувають значні вимоги до продуктивності. Недоліками цієї схеми є те, що дані можуть виявитися несумісними з точки зору користувача, тобто надмірність може виявитися на логічному рівні, а це означає, що термін надмірність, яка

контролюється, не можна застосовувати. Репліки ставатимуть несумісними через ситуації, яких складно або неможливо уникнути та наслідки яких важко виправити.

Приклад. Конфлікти можуть виникати через порядок застосування оновлень. Припустімо, що в результаті виконання транзакції *A* відбувається вставка рядка в репліку *X*, після чого транзакція *B* видаляє цей рядок, припустімо також, що *Y* — репліка *X*. Якщо оновлення розповсюджуватиметься на *Y*, але здійснюватиметься з реплікою *Y* у зворотньому порядку (наприклад, через різні затримки передачі), то транзакція *B* не знайде в *Y* рядок, який потрібно видалити, тому не виконає свою дію, після чого транзакція *A* вставить цей рядок, тобто репліка *Y* міститиме вказаний рядок, а репліка *X* — ні.

В цілому задачі усунення конфліктних ситуацій та забезпечення узгодженості реплік є вкрай складними, хоча для комерційних баз даних переважно використовують асинхронну реплікацію, яку називають керуванням копіюванням. Основна відмінність між реплікацією та керуванням копіюванням полягає у тому, що, якщо застосовують реплікацію, то оновлення однієї репліки розповсюджують на всі інші автоматично. У разі використання керування копіюванням, навпаки, не застосовують автоматичного розповсюдження оновлень. Копії даних створюються та керуються пакетним або фоновим процесом, який є віддаленим за часом від транзакцій оновлення.

Керування копіюванням загалом є більш ефективним у порівнянні з реплікацією, оскільки за один раз можна копіювати більші обсяги даних. До недоліків можно віднести те, що значну частину часу копії даних не є ідентичними базовим даним, тому користувачі мають враховувати, коли саме були синхронізовані такі дані. Зазвичай керування копіюванням спрощують завдяки вимозі, щоб оновлення застосовувались у відповідності до схеми первинної копіїї певного виду.

1.4.3. Поширення оновлень

Операції оновлення в розподілених і реплікованих сховищах даних зазвичай ініціюються клієнтом, після чого пересилаються на одну з копій. Звідти оновлення поширюється на інші копії, гарантуючи тим самим збереження несуперечливості.

Одне з важливих питань під час проектування розподіленої системи полягає в тому, яка саме інформація буде поширюватися. Існує три основних способа поширення оновлень**:**

- поширювати тільки повідомлення про оновлення;
- передавати дані з однієї копії до другої;
- поширювати операцію оновлення по всіх копіях.

Поширення повідомлення здійснюється відповідно до **протоколів про неспроможність** (*invalidation protocols*), тобто інші копії інформуються про здійснене оновлення, а також про те, які дані в них стали неправильними. За цією інформацією можна визначати, яку саме частину сховища даних було змінено. У такому разі дані не передаються, а передається тільки повідомлення. Якщо необхідно виконати будь-яку операцію з неоновленою копією, то спочатку потрібно оновити цю копію, після чого виконувати потрібну операцію, при цьому конкретні дії з оновлення залежать від підтримуваної моделі несуперечливості.

Основна перевага протоколів про неспроможність полягає в тому, що вони використовують мінімум пропускної здатності мережі. Єдина інформація, яку необхідно передавати, – це вказівка на те, які дані стали неправильними. Протоколи про неспроможність зазвичай є ефективними у разі більшої порівняно з операціями читання, кількості операцій запису, тобто в умовах, коли відношення операцій читання до операцій запису невелике.

Наприклад, сховище даних, у якому оновлення поширюють за рахунок розсилання оновлених даних усім реплікам. Якщо обсяг

оновлених даних значний, а оновлення, порівняно з операціями читання, відбуваються часто, може виникнути ситуація, коли два оновлення відбуваються одне за одним так, що у проміжку між ними операцій читання немає. Відповідно, поширення першого оновлення на всі репліки фактично не потрібне, оскільки його результати будуть стерті другим оновленням. Замість виконання операції оновлення можна надіслати повідомлення про те, що дані були оновлені, і це підвищить продуктивність системи в цілому.

Передача оновлених даних між репліками – це друга альтернатива, яка застосовується, коли відношення операцій читання до операцій запису порівняно велике. У цьому разі висока ймовірність того, що оновлення буде ефективним у тому розумінні, що модифіковані дані прочитають до того, як відбудеться наступне оновлення. Замість того, щоб поширювати оновлені дані, достатньо вести журнали оновлень і для зниження навантаження на мережу передавати тільки ці журнали. Крім того, передачу даних часто агрегують так, щоб кілька модифікацій впакувати в одне повідомлення. Це також знизить витрати на взаємодію.

Третій підхід полягає в тому, щоб відмовитися від перенесення модифікованих даних цілком, а вказувати кожній репліці, яку операцію з нею варто виконати. Цей метод, який також називають **активною реплікацією** (*active replication*), передбачає, що кожна репліка подана процесом, здатним «активно» зберігати актуальність своїх даних за рахунок виконання операцій з ними. Основний виграш від активної реплікації полягає в тому, що оновлення часто вдається поширювати з мінімальними витратами на взаємодію, зумовленими тим, що параметрів, асоційованих із певною операцією, порівняно небагато. Натомість, кожній репліці може знадобитися більша процесорна потужність, особливо якщо операції будуть складними.

Інше питання проектування полягає в тому, чи потрібно оновлення просувати (*push*) або опитувати (*pull*) сервер щодо їх появи. У

протоколах, в основі яких просування (*push-based protocols*), відомих також за назвою «серверні протоколи» (*server-based protocols*), оновлення поширюються іншими репліками, не очікуючи надходження від них запитів на одержання відновлень. Подібний підхід часто використовується у проміжку між постійними та ініційованими сервером репліками, але може застосовуватися також і для передачі оновлень у клієнтські кеші. Серверні протоколи застосовуються, коли в репліках варто підтримувати передусім порівняно високий рівень несуперечливості, тобто коли репліки мають бути ідентичними.

Вимога високого ступеня несуперечливості зумовлена тим, що постійні та ініційовані сервером репліки, так само як і великі разом використовувані кеші, часто поділяються множиною клієнтів, які здійснюють переважно операції читання. Відповідно, співвідношення операцій читання до операцій запису для кожної репліки порівняно високе. У таких випадках протоколи, які грунтуються на просуванні, ефективніші у тому розумінні, що кожне «просунуте» оновлення буде читати один або більше користувачів. Окрім того, протоколи просування роблять несуперечливі дані доступними одразу після запиту.

Натомість у підходах, в основу яких покладено **опитування** (*pull-based approach*), сервер або клієнт звертається із запитом до другого сервера, вимагаючи надіслати йому всі доступні до цього моменту оновлення. Такі методи або клієнтські протоколи (*client-based protocols*), часто використовують у процесі роботи з клієнтським кешем. Так стандартна стратегія, застосовувана під час роботи з кешами в Web, передбачає попередню перевірку актуальності кешованих даних. Коли кеш одержує запит на локально кешований елемент даних, він звертається до «рідного» web-сервера, перевіряючи, чи не був цей елемент даних модифікований після його кешування. Якщо модифікація відбулася, модифіковані дані передаються спочатку в кеш, а потім –

клієнтові, що їх запросив; якщо модифікації не було - клієнтові відразу передаються кешовані дані, тобто клієнт опитує сервер у пошуках оновлень.

Метод опитування ефективний у разі порівняно невеликого відношення кількості операцій читання до кількості операцій запису, зокрема у разі не поділюваних клієнтських кешів, з якими працює тільки один клієнт. Однак навіть якщо кеш спільно використовує множина клієнтів, підхід, який грунтується на опитуванні, може бути досить ефективним за умови, що кешовані елементи даних рідко відбувається спільний доступ. Основний недолік цієї стратегії порівняно із просуванням полягає в тому, що у разі **кеш-промаха** (*cache miss*), тобто коли дані в кеші виявляються неактуальними, час відгуку збільшується.

Порівнюючи підходи, в основу яких покладено опитування і просування, можна відзначити багато нюансів, частину яких ілюструє табл. 1.4. Для простоти обмежимося розглядом системи клієнт-сервер, що складається з одного нерозподіленого сервера й декількох клієнтських процесів, кожний з яких використовує власний кеш.

Таблиця 1.4. Порівняння протоколів опитування і просування для системи з одним сервером і декількома клієнтами

Інформація, яка розповсюджується	**Інформація, яка отримується**	
	Просування	**Опитування**
Інформація про стан на сервері	Список клієнтських реплік і кешів	Не отримується нічого
Зміст повідомлень	Оновлення (і, можливо, спочатку запит)	Запит і оновлення
Дані про час відгуку для клієнта	Негайно (або, у разі запиту, час одержання оновлення)	Час одержання оновлення

Важливим для сервера аспектом використання протоколів, які грунтуються на просуванні, є необхідність відслідковувати всі клієнтські кеші. Навіть якщо не враховувати того, що сервери із фіксацією стану зазвичай менш стійкі до збоїв, відстеження всіх

клієнтських кешів може спричинити значне навантаження на сервер. Наприклад, за таким підходом web-сервер має пам'ятати про кожен з десятків тисяч клієнтських кешів. Щоразу після відновлення вмісту web-сторінки сервер буде змушений відшукувати у своєму списку ті клієнтські кеші, у яких зберігається копія цієї сторінки, і потім передавати оновлення всім знайденим клієнтам. Більше того, якщо клієнт у пошуках вільного місця на диску втратить сторінку у своєму кеші, він повинен буде повідомити про це сервер, який вимагатиме додаткового обміну повідомленнями.

Повідомлення, якими доведеться обмінюватися клієнтові й серверу, також різні. У разі просування застосовується єдиний тип взаємодії – оновлення, передані сервером клієнтові. Якщо як оновлення клієнт отримує тільки повідомлення про неспроможність, то для одержання модифікованих даних він потребуватиме додаткової взаємодії. У разі опитування клієнт повинен опитувати сервер й, якщо буде потрібно, одержувати модифіковані дані.

І, нарешті, час відгуку для клієнта також буде різним. Коли сервер просуває модифіковані дані в кеш клієнта, зрозуміло, що час відгуку для клієнта дорівнюватиме нулю. Якщо просуваються тільки повідомлення про неспроможність, то час відгуку буде таким самим, як і у разі опитування (час відгуку визначатиметься часом, необхідним для одержання із сервера модифікованих даних).

Спроба врахувати недоліки та переваги обох підходів здійснена у змішаній формі поширення оновлень, в основі якої лежить **оренда** (*lease*), що є зобов’язанням сервера певний час постачати оновлення клієнтові. Коли строк оренди минає, клієнт запитує в сервера оновлення й, у разі потреби через опитування одержує від нього модифіковані дані. Клієнт може також після закінчення попереднього строку оренди запросити у сервера новий, щоб і далі одержувати оновлення за рахунок просування їх із сервера.

Оренду було запропоновано зі зручним механізмом динамічного перемикання між стратегіями просування й опитування. За такої умови можливо формування гнучкої системи оренди, яка дозволяє динамічно адаптувати строк оренди до різних критеріїв. Розрізняють три типи оренди (відзначимо, що в усіх випадках до закінчення терміну оренди оновлення просуваються сервером на клієнта) залежно від ознак, на яких ґрунтується оренда, зокрема таких:

- «віку» елемента даних;
- того, наскільки часто клієнт вимагає оновлення копії у своєму кеші;
- обсягу дискового простору, затрачуваного сервером на збереження стану.

Оренда, яка використовує «вік» елемента даних, закінчується для різних елементів даних згідно з часом їх останньої модифікації, якщо дані протягом тривалого часу не модифікуються, то вони не модифікуватимуться ще протягом деякого часу. Для даних у Web це припущення є доцільним. Задаючи тривалі строки оренди немодифікованих даних, можна значно знизити кількість повідомлень оновлення порівняно з однаковими строками оренди для всіх елементів даних.

Наступна ознака - частота вимог оновити копії у своєму кеші конкретним клієнтом. За такої оренди сервер установлює тривалі строки оренди тим клієнтам, кеш яким часто оновлюється, натомість клієнт, який лише час від часу запитує конкретний елемент даних, одержує на цей елемент даних короткострокову оренду. В результаті такої стратегії сервер відслідковує лише тих клієнтів, які активно користуються його даними й мають потребу у високому рівні їх несуперечливості.

Остання ознака – обсяг простору, затрачуваний сервером на збереження стану клієнтів. Коли сервер розуміє, що поступово «переповняється», він зменшує час нової оренди, надаваної клієнтам. У

результаті такого підходу серверу доводиться відслідковувати мінімум клієнтів, за рахунок чого сервер динамічно зменшує обсяг дискового простору, який відводить на збереження стану клієнтів, перетворюючись у мережі на сервер без фіксації стану. Це розвантажує його, і він починає працювати ефективніше.

З питанням про те, як розсилати оновлення – просуваючи або опитуючи – тісно пов'язане і питання про метод розсилання – цільовий або груповий. Під час **цільового розсилання** (*unicasting*) сервер є частиною сховища даних і передає оновлення на N інших серверів через відправлення N_j окремих повідомлень, по одному на кожен сервер. У разі **групового розсилання** (*multicasting*) за ефективне передавання одного повідомлення декільком одержувачам відповідає базова мережа.

Іноді дешевше використати доступні засоби групового розсилання, ніж інші методи розповсюдження оновлень, наприклад коли всі одержувачі зосереджені в межах однієї локальної мережі, в якій наявні апаратні засоби **широкомовного розсилання** (*broadcasting*). Тоді витрати на передачу одного повідомлення за рахунок широкомовного або групового розсилання не перевищують витрат на наскрізну (від точки до точки) передачу цього повідомлення. У такій ситуації передача оновлень через цільове розсилання є недоцільною.

Групове розсилання часто може бути ефективним у сполученні з поширенням оновлень за рахунок просування. У цьому разі сервер, який хоче «просунути» оновлення на кілька інших серверів, використовує одну групу групового розсилання, натомість у разі опитування оновлення копії потрібно зазвичай тільки одному серверу або одному клієнтові. За таких умов цільове розсилання – найбільш ефективне рішення.

1.4.4. Епідемічні протоколи

Для багатьох сховищ даних достатньо лише причинної несуперечливості, тобто якщо немає оновлень, можна лише гарантувати причинну ідентичність копій. Поширення оновлень у причинно-несуперечливих сховищах даних часто реалізується за допомогою класу алгоритмів, відомих за назвою **епідемічних протоколів** (*epidemic protocols*). Епідемічні алгоритми не допускають конфліктів оновлення, тому що для їх вирішення потрібно створювати додаткові індивідуальні програмні засоби, натомість епідемічні алгоритми орієнтовані на те, щоб поширити оновлення за допомогою якомога меншої кількості повідомлень. Для цього кілька оновлень часто поєднуються в одне повідомлення, яким потім обмінюються два сервери.

Основний принцип цих алгоритмів добре демонструє приклад, коли всі оновлення конкретних елементів даних ініціюються одним сервером, що виключає можливість конфліктів подвійного запису.

Використовуючи термінологію епідеміологів, сервер, що є частиною розподіленої системи, називають **інфікованим** (*infective*), якщо на ньому зберігається оновлення, яке він готовий поширювати на інші сервери. Сервер, який ще не одержав оновлення, називають **чутливим** (*susceptible*). І нарешті, сервер, дані якого було оновлено, але він не готовий або нездатний поширювати оновлення далі, називають **очищеним** (*removed*).

Одна з популярних моделей поширення – **антиентропія** (*anti-entropy*). Відповідно до цієї моделі сервер *P* випадковим чином обирає інший сервер *Q*, після чого обмінюється з ним оновленнями. Розрізняють три способи обміну оновленнями:

- сервер *P* тільки просуває свої оновлення на сервер *Q;*
- сервер *P* тільки витягає нові оновлення із сервера Q;

− сервери *P* и *Q* пересилають оновлення один одному (тобто метод «тягнучись-штовхаючи»).

Просування оновлень не задовільняє критерію «швидкість поширення оновлень» через такі причини. Передусім, відзначимо, що під час просування оновлення можуть поширюватися тільки інфікованими серверами. Однак якщо інфіковано множину серверів, імовірність того, що кожен з них випадково обере чутливий сервер, порівняно низька. Відповідно, є шанс, що якийсь із серверів протягом тривалого часу буде залишатися чутливим просто тому, що його не обрав інфікований сервер.

Натомість опитування оновлень у разі великої кількості інфікованих серверів працюватиме набагато краще, тому що поширення оновлень ініціюють чутливі сервери, тобто сам чутливий сервер має зв'язатися з інфікованим, щоб потім одержати від нього оновлення й самому стати інфікованим.

Таким чином, якщо постійно інфікований хоч один сервер, то у разі використання кожної форми антиенторпії оновлення можуть поширитися на всі сервери. Однак, щоб переконатися, що оновлення поширюються швидко, варто задіяти додатковий механізм, за допомогою якого більш-менш інфікованими одразу стануть як мінімум кілька серверів. Звичайно, в такому разі доцільно використати пряме просування оновлення на кілька серверів.

Один зі спеціальних варіантів подібного підходу є **поширенням чуток** (*rumor spreading*) або просто балаканина (*gossiping*), який працює в такий спосіб: якщо на сервері *P* є оновлений елемент даних *x*, то він зв'язується з іншим довільним сервером *Q* і намагається просунути оновлення на *Q*. Однак, можливо, *Q* уже одержав це оновлення з іншого сервера, тоді *P* може втратити інтерес до подальшого поширення оновлення, скажемо, з імовірністю 1/*k*. Інакше кажучи, він стає очищеним.

Епідемічні алгоритми є ефективними для поширення оновлень у причинно-несуперечливих сховищах даних, однак для них утрудненим є поширення видалень елементів даних. Сутність проблеми полягає в тому, що видалення елемента даних зумовлює знищення всієї інформації про цей елемент. Відповідно, якщо елемент даних просто вилучити із сервера, то сервер може, одержавши старі копії елемента даних, інтерпретувати їх як оновлення, які містять новий елемент.

Проблемним є фіксування факту видалення елемента даних у вигляді чергового оновлення і збереження відповідного запису про це, при цьому старі копії вважатимуться не новими елементами, а старими версіями, які було оновлено за допомогою операції видалення. Запис про здійснене видалення виконується через розсилання свідотств про смерть (*death certificates*). У такому разі залишається проблема, яка полягає в тому, що й свідотства про смерть також мають бути вилучені, інакше кожен сервер поступово накопичить значну за обсягом не потрібну локальну базу даних з історичною інформацією про вилучені елементи даних. Для вирішення цієї проблеми використовують **сплячі свідотства про смерть**. Кожне свідотство про смерть отримує відмітку, яка фіксує час створення. Якщо припустити, що оновлення поширюються на всі сервери за відомий кінцевий час, то після закінчення максимального часу поширення свідотство про смерть можна видаляти.

Сплячі свідотства про смерть використовують для отримання твердої гарантії того, що видалення дійсно поширилося на всі сервери, але тільки дуже небагато серверів підтримують їх, ніколи не передаючи далі. Припустімо, сервер P має таке свідотство на елемент даних x. Якщо на P надійде застаріле оновлення для x, то сервер P відреагує на цю подію повторним розсиланням свідотства про смерть цього елемента даних.

1.5. Протоколи несуперечливості

Протокол несуперечливості (*consistency protocol*) описує реалізацію конкретної моделі несуперечливості. Дуже важливі й широко використовуються моделі несуперечливості такі як послідовна несуперечливість, слабка несуперечливість зі змінними синхронізації, а також атомарні транзакції.

Протоколи послідовної несуперечливості можна класифікувати залежно від того, чи існує первинна копія даних, якій мають передаватися всі операції запису. Якщо такої первинної копії не існує, то операції запису можуть бути ініційовані кожною з реплік.

1.5.1. Протоколи на основі первинної копії

У протоколах на основі первинної копії кожен елемент даних *x*, що перебуває у сховищі даних, має асоційований з ним первинний елемент даних, відповідальний за координацію всіх операцій запису в елемент *x*. Вирізнити первинний елемент даних можна, знаючи, що він постійно перебуває на вилученому сервері, або, що після перенесення первинного елемента в той процес, у якому було ініційовано операцію запису, цей запис може виконуватися локально.

Протоколи вилученого запису. Найпростіший протокол на основі первинної копії - той, у якому всі операції читання й запису передаються на вилучений сервер (єдиний), у результаті чого дані взагалі не реплікуються, вони просто розміщуються на єдиному сервері, з якого їх неможливо куди-небудь перемістити. Ця модель традиційно використовується в системах клієнт-сервер, причому сервер може бути розподіленим.

Протоколи локального запису. Розрізняють два типи протоколів локального запису на основі первинної копії.

Перший передбачає наявність єдиної копії кожного елемента даних *x,* тобто реплік немає. Коли процес має виконати операцію з деяким елементом даних, єдина копія цього елемента передається у процес, після чого виконується операція. Цей протокол призначений переважно для повністю розподілених нереплікованих версій. Несуперечливість зберігається тому, що наявна лише одна копія кожного елемента даних. Одна з основних проблем для такого методому повного переносу — відслідковування поточного розташування кожного елемента даних. В рішеннях для локальних мереж можно використати вбудовані засоби широкомовного розсилання. Альтернативні рішення — передача вказівників та підходи з використанням бази даних. Такі рішення використовують у розподілених системах з поділюваною пам'яттю.

Однак у разі роботи з масштабними та глобальними сховищами даних необхідни інші механізми, наприклад ієрархічні служби локалізації.

Другий тип протоколу локального запису є протокол первинного архівування, в якому первинна копія переміщується між процесами, які мають виконати операцію запису. Якщо процес змінюватиме елемент даних *x,* то він знаходить його первинну копію, після чого переміщує її туди, де знаходиться сам. Основна перевага цього подходу полягає у тому, що численні послідовні операції запису виконують локально, в той час як процеси читання все-таки оперуватимуть своїми локальними копіями. Слід зазначити, що підвищення працездатності досягається тільки тоді, коли після оновлення первинної копії інші оновлення розповсюджуватимуться з використанням неблокуючого протоколу. Такий протокол використовують в багатьох розподілених системах з поділюваною пам'яттю.

Протокол первинного архівування з локальним записом також можливо використовувати для мобільних пристроїв, які здатні працювати без мережі. Перед від'єднанням мобільний пристрій стає первинним сервером для всіх елементів даних, які на ньому мають змінити. Оскільки мобільний пристрій від'єднано від мережі, він локально здійснює всі

операції оновлення даних, а всі інші процеси мають в цей час виконувати читання, але не змінювати дані. Пізніше, коли мобільний пристрій знову буде під'єднано до мережі, оновлення розповсюджуватимуться з первинного сервера на сервери резервного копіювання, завдяки чому сховище даних приводиться до несуперечливого стану.

1.5.2. Протоколи реплікованого запису

У протоколах реплікованого запису операції запису можуть здійснюватися на багатьох репліках, а не на одній, як у разі протоколів на основі первинної копії. Розрізняють протоколи з активними репліками, у яких операції передаються на всі репліки, і протоколи несуперечливості, які ґрунтуються на більшості голосів під час голосування.

Активна реплікація. Під час активної реплікації на кожній з реплік виконується асоційований з нею процес, який здійснює операції оновлення даних. На відміну від інших протоколів, оновлення зазвичай поширюються за допомогою операції запису, яка виконує оновлення даних, тобто кожній репліці надсилається операція запису. Однак, як було показано раніше, можна також надсилати й операції оновлення.

Протоколи кворуму. Інший підхід до підтримки реплікованих операцій запису грунтується на використанні **голосування** (*voting*). Основна ідея підходу полягає в тому, щоб клієнт до початку читання або запису даних запитував й одержував дозвіл на голосування у серверів.

Для читання файла, який має *N* реплік, клієнт має зібрати **кворум читання** (*read quorum*) — довільний набір N_R або більше серверів та запросити їх дозволу на оновлення. Як тільки вони погодяться, файл зміниться і з новим файлом буде асоційовано новий номер його версії. Номер версії використовують для ідентифікації версії файла, він є

однаковим для всіх оновлених файлів. Для читання реплікованого файлу клієнт також має зв'язатися з половиною серверів плюс ще з одним та запросити у них номери версій, асоційованих з файлом. Якщо всі номери версій узгоджені, серед них має бути й останняя версія, оскільки намагання здійснити оновлення тільки на серверах, яким не было направлено запит, не можна виконати, — вони не становлять більшості.

Відповідно, для модифікації файлу потрібен **кворум запису** (*write quorum*), який утворено як мінімум N_w серверами. Значення N_R та N_w мають задовільняти таким двум умовам:

$N_R + N_w > N$,

$N_w > N/2$.

Перше обмеження використовують для запобігання конфліктів читання — запису, а друге — для запобігання конфліктів подвійного запису. Тільки після того як відповідна кількість серверів дасть згоду на операцію, файл буде можливо прочитати або змінити.

1.5.3. Протоколи узгодження кешів

Кеші - особливий вид реплікації в тому розумінні, що вони зазвичай керуються клієнтами, а не серверами. Однак протоколи узгодження кешів, які відповідають за те, щоб уміст кешу відповідав серверним реплікам, не дуже відрізняються від розглянутих протоколів несуперечливості.

Розробці і реалізації кешу, особливо в контексті мультипроцесорних систем зі спільно використовуваною пам'яттю, було присвячено дуже багато праць, у яких велика кількість рішень грунтується на апаратній підтримці, наприклад, для створення можливості ефективного широкомовного розсилання або контролю. У контексті технологій проміжного рівня розподілених систем,

побудованих на засадах операційних систем загального призначення, більш розповсюджені програмні реалізації кешів, коли для класифікації протоколів кешування використовують два різні критерії.

Передусім рішення для кешів можуть розрізнятися за **стратегією виявлення узгодженості** (*coherence detection strategy*), тобто залежно від того, коли фіксується факт наявності суперечливості. У статичних рішеннях необхідний аналіз перед виконанням здійснює компілятор, визначаючи, які дані через кешування можуть насправді стати суперечливими. Компілятор вставляє інструкції в програмне забезпечення, які дозволяють уникнути суперечливості. У сучасних розподілених системах зазвичай застосовуються динамічні рішення, у яких неузгодженості виявляються під час виконання. Перевірку може, наприклад, здійснювати сервер, який визначає, чи модифікувалися дані після кешування.

У разі розподілених БД протоколи на основі динамічного виявлення узгодженості можуть бути далі класифіковані за моментом, коли саме у процесі виконання транзакції виявляється суперечливість. Виокремлюють три варіанти протоколів. Перший, коли у процесі виконання транзакції відбувається доступ до елемента даних і клієнт має потребу в підтвердженні несуперечливості цього елемента даних із тією версією, яка зберігається на сервері (можливо, реплікованому). Транзакція не може працювати з кешованною версією, поки несуперечливість останньої не буде підтверджена.

Другий, оптимістичний підхід, полягає в тому, щоб дозволити виконання транзакції під час перевірки. У цьому разі, починаючи транзакцію, вважають, що кешовані дані відповідають дійсності. Якщо пізніше буде виявлено помилку, то виконання транзакції буде перервано.

Третій підхід полягає в тому, щоб перевіряти кешовані дані тільки перед самим підтвердженням транзакції. Цей підхід є аналогічним

підходу зі схемою оптимістичного керування паралельним виконанням, коли транзакція починає працювати з кешованими даними, сподіваючись на їх коректність. Після завершення всього процесу обробки отримані дані перевіряються на несуперечливість. Якщо використовувалися неактуальні дані, то транзакція переривається.

Протоколи узгодження кеша можуть розрізнятися також і за **стратегією встановлення узгодженості** (*coherence enforcement strategy*), яка визначає, як потрібно погоджувати кеш із копіями, що зберігаються на серверах. Найпростіше рішення – взагалі заборонити кешування загальних даних, а зберігати такі дані лише на серверах, що підтримують їх несуперечливість за допомогою одного з раніше описаних протоколів первинної копії або реплікованого запису, а клієнтам дозволити кешування тільки їх внутрішніх даних. Очевидно, що таке рішення дає лише незначний виграш у продуктивності.

Для кешування загальних даних використовують два підходи, що дозволяють підтримувати погодженість кешів: у першому з них сервер під час модифікації елемента даних розсилає всім кешам повідомлення про неузгодженість; другий полягає в поширенні оновлення. Іноді в базах даних з архітектурою клієнт-сервер підтримується автоматичний вибір між розсиланням повідомлень про неузгодженість і поширенням оновлень.

Розглянемо, що відбувається, коли процес модифікує кешовані дані. Для кешів, орієнтованих тільки на читання, операції оновлення даних можуть виконуватися винятково серверами, що вимагає використання певних розподілених протоколів, які забезпечують поширення оновлень на кеші. Нерідко для цього застосовують підхід, який ґрунтується на опитуванні даних. У цьому разі, коли клієнт виявляє, що його кеш застарів, він запитує на сервері оновлення.

Альтернативний підхід полягає в тому, щоб дозволити клієнтові безпосередньо модифікувати кешовані дані й передавати оновлення на

сервери. Такий підхід вимагає використання **кешів наскрізного запису** (*write-through caches*), часто застосовуваних у розподілених файлових системах. Насправді кешування з наскрізним записом подібне протоколу локального запису на основі первинної копії, у якому клієнтський кеш є тимчасовим первинним сервером. Щоб гарантувати несуперечливість (послідовну), клієнт повинен мати виключні права на запис, інакше можливі конфлікти подвійного запису.

Кеші наскрізного запису потенційно дають максимальний виграш у продуктивності порівняно з іншими схемами, оскільки всі операції виконуються локально. Щоб пригальмувати поширення оновлень, допускають здійснення декількох операцій запису у проміжках між сеансами зв'язку із сервером. Такий підхід є підходом з використанням **кешу зворотного запису** (*write-back cache*), що також широко застосовується в розподілених файлових системах.

1.6. Висновки

1. Реплікацію даних застосовують через її дві основні переваги-підвищення надійності та продуктивності розподілених систем, але реплікація породжує проблему суперечливості: у разі оновлення однієї з реплік вона відрізняється від інших.
2. Для збереження несуперечливості реплік необхідно поширювати оновлення так, щоб тимчасова суперечливість залишалася непомітною, а це знижує продуктивність, особливо для глобальних розподілених систем.
3. Несуперечливість даних висуває досить жорсткі вимоги для своєї підтримки, для їх виконання застосовують моделі операцій читання та запису спільно використовуваних даних.

4. Моделі несуперечливості розділяють на строгу несуперечливість, послідовну несуперечливість, лінеаризуємість, причину несуперечливість та несуперечливість FIFO.

5. Строга несуперечливість передбачає, що операції читання завжди повертають останнє записане значення, але через неможливість підтримки глобального часу у розподілених системах строгу несуперечливість не реалізовано.

6. Послідовна несуперечливість та лінеаризуємість застосовують аналогічний підхід, що й у паралельному програмуванні, всі операції запису мають спостерігатися звідусіль в однаковому порядку. Лінеаризуємість є більш строгою моделлю, яка впорядковує операції у відповідності до глобального часу (з кінцевою точністю).

7. У разі причинної несуперечливості операції, потенційно пов’язані одина з другою, виконуються в порядку, який визначається цим зв’язком.

8. Несуперечливість FIFO вимагає, щоб операції одного процесу здійснювались у порядку, визначеному цим процесом.

9. Моделі слабкої несуперечливості відносять до послідовних операцій читання і запису, які передбачають, що кожна послідовність супроводжується операціями зі змінними синхронізації, такими як блокування.

10. Моделі несуперечливості, орієнтовані на клієнтські процеси, забезпечують таку поведінку системи, що під час з'єднання клієнта з новою реплікою ця репліка швидко узгоджується з даними, з якими цей клієнт працював раніше і які можуть перебувати в інших репліках.

11. У разі розповсюдження оновлень розрізняють розповсюдження повідомлень, операцій або стану, які використовують низку технологій, які по різному визначають *що, куди, у який момент* та *хто* ініціює розповсюдження оновлень.

12. Розрізняють два способи реплікації: синхронний і асинхронний. Синхронний спосіб традиційно використовують у разі, якщо допустимий час простою вимірюється у хвилинах або потрібне негайне перемикання на резервну систему під час збою. Асинхронна реплікація дозволяє здійснювати запис до основної системи зберігання даних, не чекаючи підтвердження виконання запису до резервної системи.

13. Протоколи несуперечливості описують конкретні варіанти реалізації моделей несуперечливості. Для послідовної несуперечливості та її варіантів вибір протоколу перебуває між протоколами первинної копії та реплікованого запису. У протоколах первинної копії всі операції оновлення передаються первинній копії, яка перевіряє, що оновлення правильно впорядковані й передані. У протоколах реплікованого запису оновлення одночасно передається декільком реплікам, тоді правильне впорядкування операцій ускладнено.

1.7. Запитання для самоконтролю

1. Дайте визначення поняттям «несуперечливість та реплікація». Які перваги має реплікація?
2. Реплікація об’єктів. Реплікація як метод масштабування.
3. Як побудовано моделі несуперечливості, які орієнтовані на дані?
4. Строга несуперечливість. Які вимоги висуває строга несуперечливість?
5. Лінеаризуємість та послідовна несуперечливість. Якою умовою визначається послідовна несуперечливість? Якою умовою визначається лінеаризуємість?
6. Причинна несуперечливість. Якою умовою визначається причинна несуперечливість?
7. Несуперечливість FIFO. Якою умовою визначається несуперечливість FIFO?

8. Слабка несуперечливість. Назвіть три властивості слабкої несуперечливості.

9. Вільна несуперечливість. Назвіть три вимоги, яким має задовільняти вільно несуперечливе сховище даних.

10. Опишіть два варіанти реалізації вільної несуперечливості.

11. Поелементна несуперечливість. Опишіть три умови, яким має відповідати сховище даних, що забезпечує поелементну несуперечливість.

12. Порівняйте моделі несуперечливості. Які моделі несуперечливості не потребують операцій синхронізації? Які моделі несуперечливості використовують операції синхронізації?

13. Як побудовано моделі несуперечливості, які орієнтовані на клієнтські процеси?

14. Поясніть, що таке потенційна несуперечливість, коли її використовують?

15. Поясніть, що таке монотонне читання, як воно виконується?

16. Поясніть, що таке монотонний запис, як і коли він виконується?

17. Поясніть, чим вирізняється читання особистих записів, як воно виконується?

18. Які особливості має запис за читанням, коли його застосовують?

19. Коли та для чого застосовують розміщення реплік?

20. Поширення оновлень. Які основні види поширення оновлень?

21. Як функціонують епідемічні протоколи, у якому разі їх рекомендовано використовувати?

22. Як функціонують протоколи на основі первинної копії, коли їх рекомендовано застосовувати?

23. Поясніть алгоритми роботи протоколів реплікованого запису.

24. Поясніть алгоритми роботи протоколів узгодження кешів. У чому полягає стратегія виявлення узгодженості?

25. У чому полягає стратегія встановлення узгодженості для протоколів узгодження кешів?

26. Які функціональні вимоги до серверів реплікації?

27. Наведіть класифікацію реплікацій за напрямленням реплікації.

28. Назвіть класифікацію реплікацій за часом проведення сеансу реплікації.

29. Наведіть класифікацію реплікацій за способом передачі інформації під час процесу реплікації.

30. Назвіть класифікацію реплікацій за способом аналізу реплікованої інформації.

31. Що таке синхронізація як метод реплікацій ?

32. Як здійснюється реплікація баз даних у СКБД Access?

33. Як виконується синхронізація реплік?

34. Наведіть способи й типи реплікації даних.

2. ВІДМОВОСТІЙКІСТЬ

Характерною рисою розподілених систем, яка відрізняє їх від одиничних машин, є можливість часткової відмови.

Часткова відмова відбувається у разі збою в одному з компонентів розподіленої системи і може порушити нормальну роботу деяких компонентів, проте ніяк не вплине на інші компоненти. На відміну від розподіленої системи відмова в нерозподіленій системі завжди є глобальною, тобто вона впливає на працездатність всіх її компонентів і легко може призвести до непрацездатності всього прикладного програмного забезпечення.

Створюючи розподілену систему, дуже важливо досягти того, щоб вона могла автоматично відновлюватися після часткових відмов, дещо знижуючи при цьому загальну продуктивність. Зокрема, коли б не відбувалася відмова, розподілена система у процесі оновлення має продовжувати працювати прийнятним чином, тобто бути стійкою до відмов, зберігаючи у разі відмов певний ступінь функціональності.

Атомарність – це властивість, важлива для багатьох прикладних програм. Так, у розподілених транзакціях необхідно гарантувати, щоб усі операції, які входять у транзакцію, або відбувалися, або ні.

Розглянемо, як відновлювати систему після відмов, коли і як слід зберігати стан розподіленої системи на той випадок, якщо пізніше цей стан потрібно буде відновлювати.

2.1. Поняття «відмовостійкість»

Основа всіх методик ліквідації наслідків відмов – надлишковість.

Щоб зрозуміти значення відмовостійкості в розподілених системах, спочатку необхідно з'ясувати, що для розподілених систем

означає «бути відмовостійкими». Відмовостійкість тісно пов'язана з поняттям «надійні системи» (*dependable systems*).

Надійність – це термін, що охоплює множину важливих вимог до розподілених систем, включаючи **доступність** (*availability*); **безвідмовність** (*reliability*), **безпеку** (*safety*), **ремонтопридатність** (*maintainability*).

Доступність – це властивість системи перебувати у стані готовності до роботи, яка зазвичай показує ймовірність того, що система в даний момент часу правильно працюватиме і зможе виконати свої функції, якщо користувачі того зажадають. Інакше кажучи, система з високим ступенем доступності – це така система, яка в довільний момент часу перебуває у працездатному стані.

Під **безвідмовністю** розуміють властивість системи працювати без відмов протягом тривалого часу. На відміну від доступності, безвідмовність розглядають на протязі тимчасового інтервалу, а не у певний момент часу. Система з високою безвідмовністю – це система, яка безперервно працюватиме протягом довгого часу. Між безвідмовністю і доступністю є невелика, але істотна відмінність.

Безпека визначає, наскільки катастрофічна ситуація тимчасової нездатності системи належним чином виконувати свою роботу. Так, багато систем керування процесами, які використовують, наприклад, на атомних електростанціях або космічних кораблях, повинні мати високий ступінь безпеки. Якщо керівні системи, навіть тимчасово, на короткий термін, перестануть працювати, то наслідки можуть бути жахливими. Велика кількість прикладів подій, що відбувалися у минулому, показує, як важко побудувати безпечну систему.

І, нарешті, **ремонтопридатність** визначає, наскільки складно усунути недоліки в системі. Системи з високою ремонтопридатністю можуть також мати високий ступінь доступності, особливо за наявності засобів автоматичного виявлення і усунення недоліків.

Часто надійні системи вимагають також підвищеного рівня захисту, особливо щодо несуперечливості.

Система відмовляє (*fail*), якщо вона не в змозі виконувати свою роботу. Зокрема, розподілену систему, створену для надання користувачам деяких послуг, вважатимуть такою, що перебуває у стані відмови в тому разі, якщо вона не зможе надавати всіх або деяких послуг.

Помилкою (*error*) називають такий стан системи, який може призвести до її непрацездатності. Так, у процесі передачі пакетів мережею деякі пакети, що дійшли до одержувача, будуть пошкодженими, тобто в цьому разі одержувач може невірно прочитати значення бітів (наприклад, 1 замість 0) або не зможе визначити, що пакет уже надійшов.

Причиною помилки є відмова (*fault*). Знайти причину помилки дуже важливо. Так, спричинити пошкодження пакетів може несправне або неякісне середовище передачі, за якого усунути відмову порівняно легко. Проте помилки передачі в безпроводних мережах можуть бути зумовлені, наприклад, поганою погодою. Побудова надійних систем тісно пов'язана з керуванням відмовами.

Найбільш важливим питанням є **відмовостійкість** (*fault tolerance*), під якою розуміють здатність системи надавати послуги навіть за наявності відмов.

Відмови зазвичай поділяють на швидкоплинні, переміжні й постійні.

Швидкоплинні відмови (*transient faults*) відбуваються одноразово і більше не відбуваються, тобто якщо повторити операцію, то вони не виникають.

Приклад. Якщо птах пролетів через промінь мікрохвильового передавача, то в деяких мережах це може зумовити втрату бітів. Якщо передавач після заданої паузи повторить відправлення, то ймовірно передача відбудеться без помилок.

Переміжні відмови (*intermittent faults*) з'являються і зникають, «коли захочуть», а потім з'являються знову і т. д. Через нестабільність поведінки системи у разі наявності переміжних відмов їх не завжди можна виявити.

Приклад. Якщо втрачено контакт у роз'ємі, де поєднуються окремі елементи мережі, то це може стати причиною переміжних відмов. Виявити таку відмову складно, тому що іноді контакт може з'явитися, а потім зникнути.

Постійними відмовами (*permanent faults*) називають відмови, які продовжуються доти, поки компонента, який відмовив, не буде замінено.

Приклад. Постійними відмовами можна вважати мікросхеми, які згоріли, помилки у програмному забезпеченні або головки дисків, що змістилися.

2.1.1. Моделі відмов

Система, що відмовила, не може правильно виконувати ту роботу, для якої її було створено. Якщо розглядати розподілену систему як набір серверів, які взаємодіють один з одним і з клієнтами, то некоректне виконання роботи означатиме, що сервери або (та) комунікаційні канали не в змозі виконувати свою роботу. Проте сервери, які не функціонують, не завжди спричиняють відмови системи. Якщо сервер для коректної роботи потребує послуг інших серверів, то причиною збою, можливо, є щось інше. В розподілених системах наявна значна кількість взаємозалежностей між їх елементами.

Приклад. Диск, що відмовив, ускладнює роботу файлового сервера, який розроблено для реалізації файлової системи з високим ступенем доступності. Якщо цей файловий сервер є частиною розподіленої бази даних, то під загрозою перебуває робота всієї бази, оскільки доступною є тільки частина даних.

Щоб краще розуміти, наскільки суттєвою є кожна відмова, було розроблено різні схеми класифікації.

Поломка (*crash failure*) виникає у разі раптової зупинки сервера, який до цього моменту працював нормально. Важлива особливість поломки полягає в тому, що після зупинення сервера жодних ознак його роботи не спостерігається. Типовий приклад поломки – повне зависання операційної системи, коли єдиним подоланням проблеми є перезавантаження. Багато операційних систем персональних комп'ютерів ламаються досить часто. У зв’язку з цим було виправданим перенесення кнопки «*Reset*» із задньої частини корпусу комп'ютера на передню панель.

Пропускання даних (*omission failure*) виникає у тому разі, коли сервер неправильно реагує на запити, що може бути зумовлено різними причинами. У разі пропускання приймання (*receive omission*) сервер може, наприклад, не одержувати запитів. Треба відзначити, що така помилка може відбутися, зокрема, тоді, коли з'єднання між клієнтом і сервером установлено абсолютно правильно, але на сервері не запущено процесу для приймання запитів, що надходять. Пропуск приймання зазвичай не впливає на поточний стан сервера, але сервер не має інформації про надіслані йому повідомлення.

Схожа помилка – **пропуск передачі** (*send omission*) – відбувається, коли сервер виконує свою роботу, але з яких-небудь причин не в змозі надіслати відповідь, наприклад, у разі переповнювання буфера передачі, якщо сервер не готовий. Відзначимо, що на відміну від пропуску приймання в цьому разі сервер може набути стану, який відповідає повному виконанню послуги для клієнта. Згодом, якщо виявиться, що відбувся пропуск передачі, то сервер, ймовірно, має бути готовий до того, що клієнт повторно надішле свій останній запит. Інші типи пропусків не відносяться до процесів взаємодії і можуть бути зумовлені помилками у програмі, зокрема нескінченними циклами або некоректною роботою з пам'яттю, здатними спричинити «зависання» сервера.

Наступний клас помилок – це помилки синхронізації. **Помилки синхронізації** (*timing failures*) виникають у разі очікування відповіді довше певного тимчасового інтервалу. Проте часто сервер відповідає надто пізно, тоді вважають, що відбулася **помилка продуктивності** (*performance failure*).

Ще один важливий тип помилок – **помилки відгуку** (*response failures*), за яких відповіді сервера просто неправільні. Розрізняють два типи помилок відгуку: помилки значення та помилки передачі стану.

У разі **помилки значення** (*value failure*) сервер дає неправільну відповідь на запит, наприклад, пошукова машина систематично повертає адреси *web*-сторінок, не пов'язаних із запитом користувача.

Помилки передачі стану (*state transition failures*) характеризуються реакцією на запит, яка не відповідає очікуванням. Так, якщо сервер одержує повідомлення, яке він не в змозі розпізнати, і ніяких заходів з обробки таких повідомлень не передбачено, виникає помилка передачі стану. Зокрема, сервер може виконати за замовчанням якісь непотрібні дії.

Значно суттєвішими є **довільні помилки** (*arbitrary failures*), які називають візантійськими (*Byzantine failures*). Наприклад, сервер генерує повідомлення, які він не повинен генерувати, але система не розпізнає їх як некоректні. Неправильно функціонуючи, сервер може, беручи участь у роботі групи серверів, зумовлювати появу завідомо неправильних відповідей.

Поломка – найбільш поширена причина зупинення сервера, які ще називають помилками **аварійного зупинення** (*fail-stop failures*). Насправді, аварійно зупинений сервер просто припиняє генерувати вихідні повідомлення. За цією ознакою його зупинення виявляють інші процеси.

Зрозуміло, в реальному житті сервери, зупиняючись унаслідок пропуску даних або поломок, не оповіщають про зупинення, яке

відбудеться, тому інші процеси мають самі виявити «передчасну загибель» сервера. Проте в таких системах **зупинення без повідомлення** (*fail-silent systems*) інші процеси можуть зробити неправильний висновок про зупинку сервера. Сервер може просто повільно працювати, тобто фактично відбуватиметься помилка продуктивності.

Крім того, можливо, що сервер дає випадкові повідомлення, які інші процеси вважають сміттям. Такі помилки називають **безпечними** (*fail-safe*).

2.1.2. Маскування помилок за допомогою надлишковості

Відмовостійка система має маскувати факти помилок від інших процесів. Основний метод маскування помилок – використання **надлишковості** (*redundancy*), зокрема **інформаційної**, **тимчасової** і **фізичної** надлишковості. У разі інформаційної надлишковості до повідомлення додаються додаткові біти, за допомогою яких можна виправити збійні біти, наприклад, можна додати до даних, що передаються, код Хеммінга для відновлення сигналу в разі зашумленого каналу передачі.

У разі тимчасової надлишковості вже виконана дія, якщо це потрібно, виконується повторно. Як приклад розглянемо транзакції.

Приклад. Розглянемо транзакції: якщо транзакцію було перервано, то її можна безсумнівно повторити.

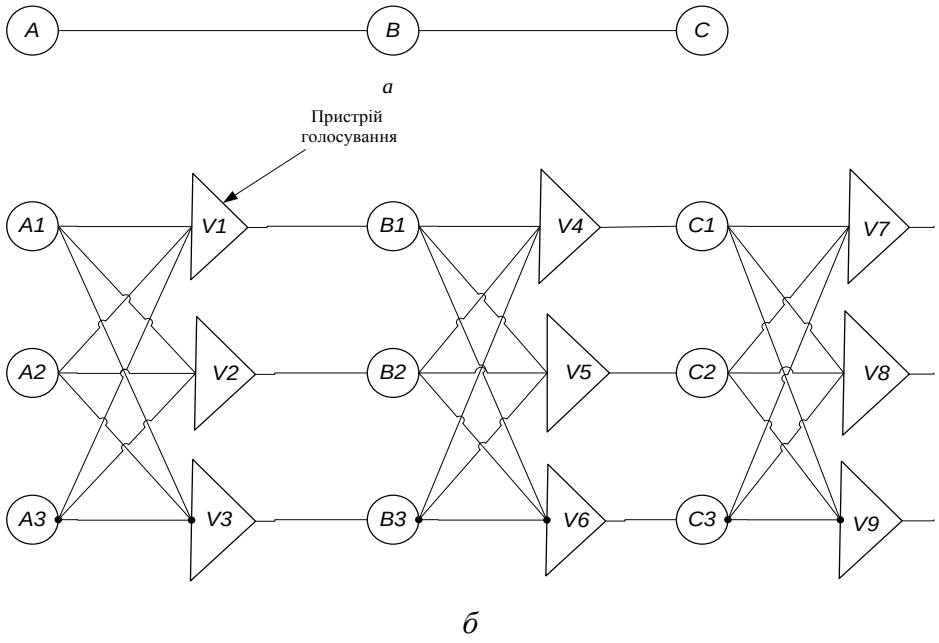
Тимчасова надлишковість допомогає добре маскувати відмови у разі виникнення прохідної або переміжної відмови.

У разі фізичної надлишковості до системи долучають додаткове устаткування або процеси, які роблять можливою роботу системи у разі втрати або непрацездатності деяких компонентів. Фізична надлишковість може бути як апаратною, так і програмною. Так, за

рахунок залучення до системи додаткових процесів, навіть у разі повної непрацездатності деяких із них система функціонуватиме правильно. Інакше кажучи, за допомогою реплікації досягається високий ступінь відмовостійкості.

Фізична надлишковість – це найбільш поширений спосіб досягнення відмовостійкості, зокрема в радіосхемах.

Приклад. Розглянемо схему, показану на рис. 2.1, *а*, на якій сигнал проходить послідовно через пристрої *A, B* і *C*. Якщо один з них несправний, результат, імовірно, буде невірним.



а

б

Рис. 2.1. Потрійне модульне резервування

На схемі кожний пристрій дублюється тричі (рис. 2.1,б). Рішення щодо переходу до наступної ділянки схеми приймається потрійним голосуванням. Пристрій голосування – це схема з трьома входами і одним виходом. Якщо два або три вхідні сигнали збігаються, то вихідний сигнал дорівнює вхідному; якщо ж усі три вхідні сигнали різні, то вихідний сигнал є не визначеним. Таку схему називають потрійним модульним резервуванням (*Triple Modular Redundancy, TMR*).

Припустімо, елемент *A2* відмовив, тоді кожний пристрій голосування *V1*, *V2* і *V3* одержує два правильні (ідентичні) вхідні сигнали і один неправильний, і кожен з них передає на другу ділянку ланцюга правильне значення. У результаті цього ефект відмови *A2* стає повністю замаскованим, а отже вхідні сигнали елементів *B1*, *B2* і *B3* такі, неначебто ніякої відмови не відбулося.

Розглянемо тепер, що буде, якщо разом з *A2* відмовлять також елементи *B3* і *C1*. Ефект їх відмови також буде замаскованим, і всі три вихідних сигнали виявляться правильними.

На кожному етапі потрібно використовувати три пристрої голосування, тому що пристрій голосування – це компонент, який також може відмовити. Розглянемо, наприклад, відмову *V1*. Вхідний сигнал *B1* в цьому разі буде неправильним доти, поки всі інші пристрої працюють, *B2* і *B3* даватимуть однакові вихідні сигнали, і пристрої *V4, V5* і *V6* утворюють правильний результат для третього етапу. Відмова *V1* майже не відрізнятиметься від відмови *B1*, оскільки в обох випадках вихідний сигнал від *B1* буде неправильним, а під час голосування цей сигнал опиниться в меншості.

Хоча не всі відмовостійкі розподілені системи використовують технологію *TMR*, ця технологія є дуже поширеною і допомагає ясніше зрозуміти, що таке відмовостійка система і чим вона відрізняється від системи, яка складається з високонадійних компонентів, але не має відмовостійкої структури. Зрозуміло, *TMR* можна застосовувати і рекурсивно, наприклад, підвищувати надійність мікросхем, вбудувавши в них механізми *TMR*.

2.2. Відмовостійкість процесів

2.2.1. Об'єднання ідентичних процесів у групу

Для запобігання негативним наслідкам, які зумовили відмови процесів декілька ідентичних процесів об'єднують у групу, причому динамічно можуть створюватися нові групи, а також ліквідуватися старі. Під час системної операції реорганізації групи процес може приєднатися до групи або залишити її. Процесу дозволено брати участь у декількох групах одночасно, тому необхідні механізми для керування групами і членством у них.

Основна властивість усіх таких груп полягає в тому, що коли повідомлення надсилається групі, його одержують усі члени цієї групи.

Таким чином, якщо один процес групи перестане працювати, то можна сподіватися, що замість нього працюватиме другий.

Мета угрупування полягає в тому, щоб перейти від розгляду окремих процесів до розгляду нової абстракції – групи процесів. Так, процес може надсилати повідомлення групі серверів, не знаючи нічого про те, скільки їх і де вони містяться, причому склад групи серверів під час кожного виклику може бути різним.

Однорангові та ієрархічні групи. Всі групи поділяють відповідно до їх внутрішньої структури, серед яких розрізняють дві найбільш поширені моделі. Першу модель групи процесів називають **одноранговою**, коли всі процеси рівні між собою, тобто ніяких «керівників» немає, і всі рішення ухвалюють члени групи колективно. Другу модель, за якої в групах існує певна ієрархія процесів називають **ієрархічною**, коли один з процесів – координатор, а всі інші – прості виконавці. У такій моделі з появою запиту, створеного десь поза процесом або одним із внутрішніх робочих процесів, запит надсилається координатору, котрий вирішує, який з виконавців найефективніше обробить запит, і передає йому цей запит (рис. 2.2). Друга модель може використовувати й більш складні ієрархічні алгоритми організації групової обробки запитів.

Будь-яка із цих моделей організації обробки запитів групою має свої переваги і недоліки. Однорангова група є симетричною і не має однієї точки відмови. Якщо в одному з процесів виявлено помилку, то група просто зменшується, але продовжує існувати. Недолік однорангових груп полягає в тому, що вони мають складніший процес ухвалення рішень, тобто щоб домовитися про щось, необхідно голосувати, що спричиняє певну затримку і необхідність додаткових дій.

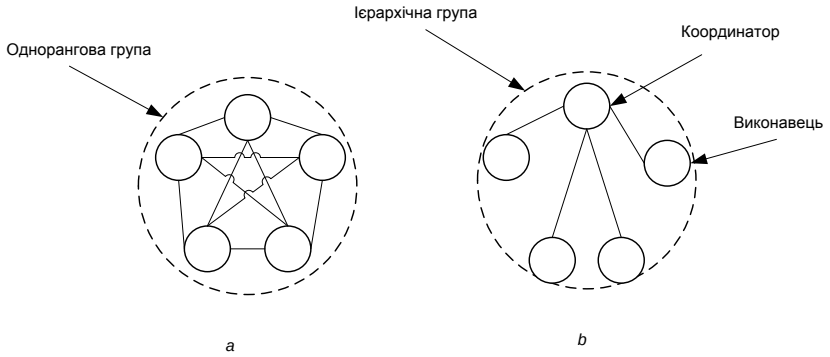


Рис. 2.2. Взаємодія в одноранговій (*а*) та ієрархічній (*б*) групах

Ієрархічна група має протилежні властивості: втрата координатора зумовлює зупинення роботи всієї групи, але поки він перебуває в робочому стані, то ухвалює рішення самостійно без участі інших членів групи.

Членство у групі. Під час групової взаємодії необхідні спеціальні методи створення і знищення груп, а також додавання процесів у групу та їх видалення з неї. Один з можливих методів – створення сервера груп (*group server*), до якого мають прямувати відповідні запити. Сервер груп підтримуватиме повну базу даних за групами і членством у них. Цей метод простий, ефективний і легко реалізовний. На жаль, він має характерний основний недолік усіх централізованих методів – єдина точка відмови. Якщо сервер груп вийде з ладу, то керування групами буде втрачено. Ймовірно, велику частину груп вдасться відтворити з працездатних членів групи, але при цьому буде повністю перервана діяльність, яка в них відбувалася.

Інший можливий метод – розподілене керування членством, тобто якщо доступна надійна групова розсилка, то зовнішній учасник взаємодії може надіслати повідомлення всім членам групи, оголосивши про своє бажання ввійти до складу цієї групи.

Для того, щоб залишити групу, її члену достатньо розіслати всім членам групи «прощальний лист», що є реалізуємим способом оголосити про добровільне припинення членства у групі, але немає способу оголосити про аварійну втрату одного із членів, тому що у

контексті відмовостійкості використання семантики аварійної зупинки заборонено. Інші члени групи мають визначити це експериментально, відзначивши, що якийсь член групи тривалий час нікому не відповідає. Як тільки стає зрозумілим, що член групи дійсно перебуває в неробочому стані (а не просто повільно працює), його слід видалити з групи.

Ще одна суттєва проблема полягає в тому, що видалення і додавання членів мають відбуватися синхронно з обробкою повідомлень з даними. Починаючи з моменту, коли процес додано до групи, він повинен одержувати всі повідомлення, які отримує група. Відповідно, коли процес залишає групу, він більше не одержує надісланих до групи повідомлень, а інші члени групи - повідомлень від нього. Для гарантування потрапляння видалення або додавання процесу до групи в правильне місце потоку повідомлень таку операцію конвертують в послідовність повідомлень, які надсилаються всій групі.

У разі відмови відразу декількох машин, через що група не може продовжувати функціонувати, використовують протокол повторного збирання групи, за якого який-небудь процес має проявити ініціативу, щоб стати координатором та відновити функціонування групи.

2.2.2. Маскування помилок та реплікація

Групи процесів пропонують вирішення частини завдання побудови відмовостійких систем. Зокрема, група ідентичних процесів дозволяє замаскувати наявність у цій групі одного або декількох процесів, що відмовили, тобто можна реплікувати процеси та організувати їх у групу, замінюючи єдиний (уразливий) процес відмовостійкою групою. Виокремлюють два способи здійснення такої реплікації – з використанням протоколів на основі первинної копії або протоколів реплікованого запису.

У разі відмовостійкості реплікація на основі первинної копії зазвичай застосовується у формі протоколу первинної архівації, коли група процесів організовується в ієрархію, в якій первинна копія координує всі операції запису. На практиці первинна копія фіксована, хоча у разі потреби її роль може узяти на себе одна з архівних копій. Насправді, якщо знайдено помилку в первинній копії, архівні копії, використовуючи певний алгоритм голосування, обирають нову первинну копію.

Протоколи реплікованого запису використовуються у формі активної реплікації або протоколів на основі кворуму. Ці рішення застосовуються для організації набору ідентичних процесів у однорангову групу. Її головна перевага полягає в тому, що така група не має єдиної точки відмови, але потребує розподіленої координації.

Ефективність використання груп процесів для підвищення відмовостійкості залежить від кількості реплікацій. Розглянемо як приклад систему реплікованого запису, яку називатимемо стійкою до k відмов, якщо вона залишиться працездатною після відмови k компонентів. Якщо компоненти, або процеси, зупиняються без повідомлення, то наявності $k + 1$ процесів достатньо, щоб забезпечити стійкість до k відмов. Якщо k з них просто припинять роботу, то використовуватиметься відповідь від одного, що залишився.

Натомість, якщо процес, у якому виникла «візантійська» помилка, продовжує працювати, розсилаючи неправильні або просто випадкові повідомлення, то для того, щоб забезпечити стійкість до відмов, необхідно як мінімум $2k+1$ процесів. За найгірших обставин помилки у процесах можуть випадково (або навіть навмисно) надати однакові результати, як й інші $k+1$ процеси, тож клієнт або пристрій голосування зможуть все ж таки повірити більшості.

Теоретично можна вважати, що система є стійкою до k відмов тому, що $k+1$ однакових результатів перевищують k однакових

результатів, але на практиці важко уявити собі обставини, за яких *k* процесів можуть помилятися, а *k+1* процесів – не можуть, тому навіть у відмовостійкій системі проводять моніторинг для перевірки коретності її функціонування.

2.2.3. Відмовостійкість програмного забезпечення

Гарантованість працездатності програмного забезпечення визначають як надскладну властивість системи гарантовано надавати задані специфікацією послуги (виконувати функції), яким можна виправдано довіряти; гарантоздатною є система, якій притаманна дана властивість. Програмне забезпечення сучасних розподілених систем, працездатність яких гарантується, грунтується на готовності, безвідмовності, функціональній безпеці, цілісності й обслуговуванні. У той же час, воно має відповідати вимогам безпеки. Безпека - комбінація конфіденційності (попередження неправомірного розкриття інформації), цілісності (попередження неправомірного правлення або стирання інформації) та готовності (попередження неправомірної відмови постачання інформації). Показники гарантованості працездатності й безпеки слід розглядати як різні аспекти загальної проблеми, тобто до них можна застосовувати загальні рішення.

Отже, підходи, що грунтуються на принципах відмовостійкості, можуть забезпечити високий рівень як безвідмовності, так і безпеки.

Способи реалізації відмовостійкості програмного забезпечення. Відмовостійкість (стійкість до відмов) - фундаментальний метод для досягнення гарантованості виконання обчислень. Способи забезпечення відмовостійкості такі: виявлення помилки; обробка несправності; аналіз пошкодження; відновлення після помилки.

Всі ці способи реалізуються за допомогою захисної надлишковості алгоритмічних, функціональних, часових та інших елементів архітектури системи з урахуванням обмежень, які визначають несправності, а також зміни вимог і умов використання.

Виявлення помилки здійснюється за допомогою перевірки правильності виконання завдання. В ідеалі правильність роботи системи має грунтуватися на перевірці того, чи відповідає ця робота специфікації системи. Таку перевірку мають виконувати незалежні від системи механізми (блоки, модулі), а специфікацію - виражати в термінах інформації, яка є зовнішньою по відношенню до системи. Однією з головних особливостей відмовостійкості програмного забезпечення є можливість перевірки правильності результату до того, як результат вийде за межі системи (блока, модуля).

Залежно від обмежень вартості й продуктивності системи перевірка може виконуватися порівнянням таких результатів:

- повторного використання системи;
- використання кількох копій однієї системи;
- використання кількох різних версій системи.

Іноді застосовують перевірку зворотним ходом: результат, який надає система, обробляється у зворотному порядку, щоб отримати відповідні вхідні дані й порівняти їх із фактичними.

Ще один спосіб – перевірка результату на відповідність обмеженням предметної області. Використовують також відсутність відповіді компоненти на повідомлення в межах визначеного часу, порівняння контрольних сум інформаційних блоків тощо.

Обробка несправності полягає у визначенні місцеположення несправної компоненти програмного забезпечення та її заміні, для цього система повинна мати у своєму розпорядженні надлишкові компоненти програмного забезпечення та резерви часу. Надлишковість у системі може бути маскувальна або динамічна. Маскувальна

надлишковість – статична, наприклад, трикратне модульне резервування (*TMR*). Динамічна надлишковість – це внутрішня надмірність компоненти, яка використовується для знаходження помилкової інформації та має бути доповнена зовнішньою надлишковістю, щоб забезпечити відновлення компоненти після усунення помилки. Відновлювання виконують за допомогою заміщення або ре конфігурації: у разі заміщення замість несправної компоненти використовується резервна компонента, яка не була активною; у разі реконфігурації функції несправної компоненти перекладаються на ті компоненти системи, які працюють успішно, при цьому загальна продуктивність системи дещо знижується, що слід ураховувати, проектуючи критичні системи. На відміну від апаратного заміщення у програмному забезпеченні резервна компонента замінює основну лише на деякий час (для певної комбінації вхідних даних), після чого система намагається повернути до роботи основну компоненту.

Для забезпечення гарантованості працездатності система або її компоненти повинні мати механізми обмеження помилки, що дозволять заблокувати процес поширення пошкодженої інформації до взаємодіючих зовнішніх систем (компонент). Після цього слід визначити ступінь пошкодження та можливі наслідки для оточуючого програмно-апаратного середовища і задіяти необхідні механізми, щоб уникнути негативного розвитку подій та аварії.

Аналіз пошкодження, зумовленого несправністю, визначає, які процеси і починаючи з якого моменту слід уважати неправильними, щоб повторно їх перезапустити. Такий аналіз виконується за допомогою методу структурування системи, що забезпечує подання роботи системи у вигляді елементарних дій. Значно простіше виконувати аналіз, починаючи з правильної контрольної точки, зафіксованої в пам’яті перед збоєм, повторно перезапустити процеси

або, враховуючи наявний резерв реального часу, замінити їх процесами, які дублюють дану функцію.

Відновлення після помилки здійснюється за допомогою двох основних методів: ретроспективного відновлення після помилки та відновлення без повернення до попереднього стану.

Ретроспективне відновлення після помилки грунтується на фіксації в пам'яті контрольних точок. Цей метод допускає незнання точного місцеположення несправності й наслідків її дії в системі. Блок відновлення містить деяку кількість резервних альтернативних алгоритмів (які називають «замінами»), на яких перезапускаються потрібні процеси з даними, взятими з пам'яті контрольних точок. Отже, ідея методу дуже проста, оскільки таке відновлення не потребує діагностування та з'ясування місцеположення несправності, але при цьому втрачається час уже виконаної роботи.

Відновлення без повернення до попереднього стану передбачає точну ідентифікацію несправності й забезпечення механізму для її усунення, при цьому заново виконуються лише ті процеси, які були виконані неправильно, тобто цей метод більш ефективний, ніж попередній. Однак він прийнятний лише для простих несправностей та механізмів їх усунення.

***N*–версійне програмне забезпечення.** Найбільшу загрозу для сучасних обчислювальних систем становлять несправності проектування. Різке підвищення складності програмного забезпечення призвело до того, що стало фактично неможливим виявляти та видаляти всі несправності проектування програмного забезпечення до введення системи у фазу використання.

Проблему несправностей проектування програмного забезпечення пом'якшують за допомогою принципу різноманітності проектів, що виник як аналог дублювання апаратних засобів. Разом з тим, в апаратурі застосовується просте копіювання каналів обчислення,

оскільки воно є ефективним засобом маскування випадкових фізичних несправностей апаратних засобів, проте у разі копіювання блока програмного забезпечення з ним будуть копіюватися і невиявлені в ньому несправності проектування, тому кожен блок програмного забезпечення, що дублює виконання однієї функції, має бути розроблений незалежно від інших, що і є сутністю концепції різноманітності проектування програмного забезпечення або багатоверсійних обчислень.

2.3. Надійний зв'язок клієнт – сервер

Здебільшого описуючи питання відмовостійкості в розподілених системах, основну увагу приділяють дефектним процесам, проте необхідно також ураховувати і дефекти взаємодії. Більшість із розглянутих моделей відмов однаково успішно можна застосовувати і до каналів зв'язку. Так, у каналах зв'язку можуть виникати поломки, пропуски, помилки синхронізації та довільні помилки. На практиці, будуючи надійні канали зв'язку, основні зусилля спрямовують на маскування поломок і пропусків. Довільні помилки часто мають вигляд повторюваних повідомлень, які виникають у результаті того, що в комп'ютерних мережах повідомлення можуть надовго «застрявати» у проміжних буферах і знов потрапляти в мережу вже після того, як початковий відправник здійснив повторне відправлення того ж повідомлення.

2.3.1. Передача «точка-точка»

У багатьох розподілених мережах надійна наскрізна передача «точка–точка» (*point-to-point*) реалізується за рахунок використання надійного транспортного протоколу, зокрема TCP (*Transmission Control*

Protocol), який маскує пропуски у вигляді втрати повідомлень, за допомогою механізму підтверджень і повторного надсилання. Ці помилки залишаються абсолютно непоміченими клієнтом TCP. Проте поломки (втрати) зв'язку часто неможливо замаскувати. Поломка може відбутися, коли з певних причин з'єднання TCP раптово уривається і повідомлення по цьому каналу більше передаватися не можуть. У багатьох випадках клієнт оповіщається про поломку каналу шляхом збудження виключення. Єдиний спосіб замаскувати помилки такого роду полягає в тому, щоб дозволити розподіленій системі автоматично встановити нове з'єднання.

2.3.2. Семантика віддаленого виклику процедур за наявності помилок

Потрібно розглянути детальніше взаємодію між клієнтом і сервером з використанням високорівневих комунікаційних механізмів, зокрема віддаленого виклику процедур (*Remote Procedure Call*, RPC) або віддаленого звернення до методів (*Remote Method Invocation*, RMI).

Призначення RPC – приховати сам факт взаємодії за допомогою викликів віддалених процедур, які виглядають так само, як локальні виклики. Якщо клієнт і сервер працюють без помилок, то механізм RPC функціонує ефективно, але у разі виникнення помилки, приховати різницю між локальними і віддаленими викликами набагато складніше.

Помилки, які можуть виникнути в системах RPC, поділяють на п'ять класів таким чином:

- клієнт не в змозі виявити сервер;
- втрата повідомлення із запитом від клієнта до сервера;
- поломка сервера після отримання запиту;
- втрата повідомлення у відповідь від сервера до клієнта;
- поломка клієнта після отримання відповіді.

Помилки кожного з цих класів ставлять різні задачі, які вимагають різних способів розв’язання.

Клієнт не може виявити сервер. Якщо клієнт не може виявити відповідний сервер, який, наприклад, вимкнено, а клієнтське програмне забепечення може бути скомпільовано з якоюсь конкретною версією клієнтської заглушки, то у разі, коли виконуваний файл тривалий час не використовується, а сервер за цей час отримав нову версію інтерфейсу та були створені й запущені нові заглушки, після запуску клієнта він не зможе відповісти серверу, бо викликає повідомлення про помилку. Оскільки такий механізм використовується для захисту клієнта від спроб обміну з «неправильним» сервером (сервером, з яким клієнт не може домовитися про використовувані параметри або вирішувані задачі), потрібно визначитися, що робити з такою помилкою, для чого змусити помилку викликати обробку *виключень* (*exception*) або використати *обробників сигналів*.

Цей підхід має свої недоліки: не всі мови підтримують такі конструкції, як винятки або сигнали; написання процедури обробки виключень або сигналу суперечить вимозі прозорості системи.

Втрата повідомлень із запитом. Для запобігання такої помилки операційна система або клієнтська заглушка, надсилаючи повідомлення, мають вмикати таймер. Якщо таймер переповнено, а відповідь або підтвердження так і не буде отримано, то повідомлення надсилається повторно. У разі, коли повідомлення дійсно зникло, сервер не побачить різниці між повторно надісланим повідомленням і оригіналом, використовуючи повторне повідомлення як оригінал. Якщо пропаде значна кількість повідомлень, то клієнт відмовиться від взаємодії з сервером, помилково вирішивши, що сервер не працює. Якщо запит не втрачено остаточно, то слід зробити так, щоб сервер зміг зрозуміти, що відбулося його повторне надсилання.

Поломка сервера. Поломкою сервера вважають такі ситуації:

- нормальну послідовність подій на сервері, коли запит надходить, обробляється і надсилається відповідь (рис. 2.5, а);

- запит надходить і обробляється, але на сервері відбувається поломка до того, як він встигає надіслати у відповідь повідомлення (рис. 2.5, б);

- запит надходить, але сервер ламається ще до початку обробки (рис. 2.5, в).

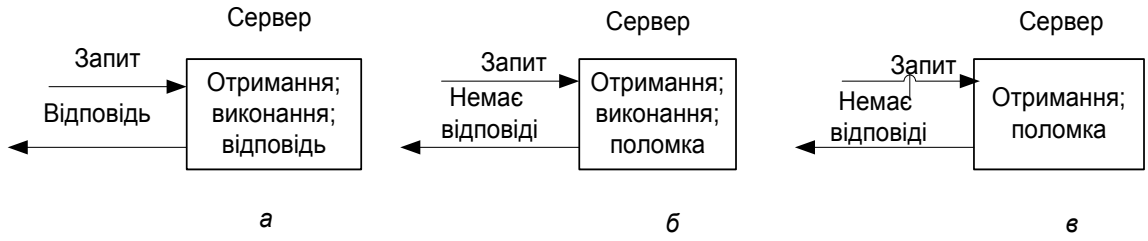


Рис. 2.5. Взаємодія сервера і клієнта: нормальна робота (*а*), поломка після обробки запиту (*б*), поломка до обробки запиту (*в*)

Правильні дії сервера у разі поломок, зображених на рис. 2.5 (*б*) і рис. 2.5 (*в*) різні. У разі (*б*) система повинна передати клієнту повідомлення про помилку (наприклад, ініціювати виклик обробника виключень), тоді як у разі (*в*) вона може просто надіслати запит повторно. Операційна система клієнта не може визначити, що саме відбулося, бо бачить тільки перепонення таймеру, а цього недостатньо, щоб діагностувати помилку.

Дану проблему вирішують по-різному. Перший підхід полягає в тому, щоб чекати перезавантаження сервера (або зв'язатися з новим сервером) і повторювати операцію до доти, поки сервер не надасть відповіді, яка дійде до клієнта. Такий прийом називають *семантикою «мінімум одного разу»* (*at least once semantics*), яка гарантує, що виклик RPC буде здійснено як мінімум одноразово, а можливо і більше.

Другий підхід, який називають *семантикою «максимум одного разу»* (*at most once semantics*), полягає в тому, щоб негайно відмовитися від подальших спроб надсилати повідомлення і повернути

повідомлення про помилку. Він гарантує, що виклик RPC буде здійснено максимум один раз, а можливо і жодного разу.

Третій підхід не гарантує діагностування та усунення помилок. Коли на сервері відбувається поломка, клієнт не одержує допомоги і жодних гарантій. Виклик *RPC* може не відбутися жодного разу або може бути виконаним будь-яку кількість разів. Основна перевага цього підходу– простота реалізації.

Отже, можливість поломки сервера радикально впливає на механізм роботи *RPC*, який є різним для однопроцесорних і розподілених систем. У разі однопроцесорної системи під поломкою сервера розуміють поломку клієнта, причому його відновлення, якщо не працює сервер, непотрібно, а у разі розподіленої системи можна і потрібно відновлювати сервер.

Втрата повідомлення з відповіддю. Для запобігання цієї помилки операційна система клієнта також має використовувати таймер. Якщо за визначений час не було одержано відповіді, то можна надіслати запит ще раз, але клієнт не буде впевнений у тому, чому немає відповіді.

Деякі операції можна повторювати стільки разів, скільки потрібно, що не зумовить ніяких порушень. Так запит на читання перших 1024 байтів файлу не має побічних ефектів і може здійснюватися так часто, як це необхідно. Запит, який має такі властивості називають *ідемпотентним* (*idempotent*).

Приклад. Розглянемо запит до банківського сервера, що вимагає переказу мільйона доларів з одного рахунка на другий. Якщо запит надійшов і був виконаний, але у відповідь повідомлення загубилося, то клієнт нічого не знає про виконання запиту і надсилає його повторно. Банківський сервер уважає, що це новий запит і виконує його, тобто здійснюється переказ двох мільйонів доларів. Отже, переказ коштів не є ідемпотентною операцією, тому для запобігання помилок під час виконання таких операцій необхідно програмувати запити так, щоб вони були ідемпотентними.

Багато запитів (наприклад, переказ коштів) не ідемпотентні за своєю природою, це потребує підходу, коли клієнт надає кожному запиту послідовний номер. Якщо сервер зберігатиме номер останнього отриманого повідомлення кожного з клієнтів, які працювали із цим сервером, то він зможе виявити відмінність між оригінальним і повторним запитами. Тоді сервер відмовиться виконувати запит удруге, але зможе повторно надіслати клієнту відповідь. Відзначимо, що такий підхід вимагає, щоб сервер відстежував усіх клієнтів. Додатковим захистом буде спеціальний біт у заголовку повідомлення, що дозволяє відрізняти початкові запити від повторного передавання.

Поломка клієнта. Якщо обчислення будуть виконані, але не виявиться замовника, який чекає результату, то такі обчислення називають *«сиротами»* (*orphans*).

Сироти зумовлюють низку проблем: спричиняють зайву витрату процесорного часу; блокують файли або якось інакше захоплюють потрібні ресурси; після перезавантаження клієнта і повторного виклику RPC замість процесу, який має відповісти, клієнт може отримати відповідь від процесу-сироти.

Використовують чотири рішення для запобігання негативним наслідкам, що викликані появою сиріт.

1. *Винищування сиріт (extermination).* Перш ніж клієнтська заглушка надішле виклик *RPC,* вона створює запис у журналі з описом того, що відбувається. Журнал зберігається на диску або іншому пристрої довготривалого зберігання інформації, здатному не втратити його після перезавантаження, після якого журнали перевіряються і всі сироти знищуються. Недоліки цього сценарію полягають у тому, що він вимагає для кожного виклику *RPC* запису на диск величезного обсягу інформації. Крім того, він може і не спрацювати, якщо сироти самі здатні робити виклики *RPC,* створюючи внучатих сиріт (*grandorphans*) або сиріт ще більшого ступеня, яких важко або взагалі

неможливо виявити. Ще одне ускладнення виникає через мережу, яка може бути розділена на фрагменти, наприклад, зламаним шлюзом, що унеможливить видалення сиріт, навіть якщо їх вдасться знайти.

2. *Реінкарнація (reincarnation).* Не записуючи інформацію на диск, розбивають час на послідовно пронумеровані інтервали перезавантаження клієнта, який за рахунок широкомовної розсилки, відправляє всім машинам повідомлення, оголошуючи про початок нового інтервалу. Коли таке повідомлення надходить на сервер, всі видалені обчислення, які було замовлено цим клієнтом, припиняються. Зрозуміло, якщо мережа поділена на частини, то можуть залишитися деякі сироти, але у відповідях від них міститиметься номер минулого інтервалу часу, що і дозволяє їх виявити.

3. *М'яка реінкарнація (gentle reincarnation).* Коли надходить повідомлення про зміну інтервалів часу, то кожна машина перевіряє, чи відбуваються на ній які-небудь видалені обчислення, і якщо так, намагається знайти їх власника. Обчислення припиняються лише в тому разі, якщо власника знайти не вдалося.

4. *Закінчення терміну (expiration).* Для кожного виклика RPC визначається стандартна тривалість роботи T. Якщо виклик не закінчив роботу, то вимагатиме наступного терміну, що заважає виявленню сиріт. Натомість, якщо після поломки клієнта до його перезавантаження проходить час T, то всіх сиріт вже точно видалено. Проблема розв'язується за рахунок вибору прийнятного часу T для викликів RPC зі значно відмінними вимогами.

Для практичного застосування непридатний жоден із цих методів, тому що видалення сиріт може призвести до непередбачуваних наслідків. Припустімо, що процес-сирота заблокував один (або більше) файл або запис бази даних. Якщо сироту раптово видалити, то блокування збережеться. Крім того, сирота може виконати низку видалених на даний час запитів на запуск інших процесів. Таким

чином, при видалення процесу-сироти може стати неможливим усунення всіх наслідки його функціонування.

2.4. Надійна групова розсилка

2.4.1. Основні схеми надійної групової розсилки

Хоча більшість систем транспортного рівня забезпечують надійні *крізні канали, засобів для надійної взаємодії з набором процесів не має.* Системи транспортного рівня дозволяють будь-якому процесу створити крізне з'єднання з будь-яким іншим процесом, з яким йому необхідно зв'язатися. Очевидно, що подібна організація зв'язку не дуже ефективна, оскільки вона вимагає значної витрати пропускної здатності мережі. Проте, якщо кількість процесів невелика, надійна групова розсилка через декілька надійних крізних каналів є найпростішим рішенням, яке застосовують на практиці.

Надійна групова розсилка (*reliable multicasting*). Повідомлення, відправлене групі процесів, має бути гарантовано доставлено всім членам цієї групи.

Щоб досягти такого рівня надійності, слід розмежувати надійний зв'язок за наявності помилково функціонуючих процесів і надійний зв'язок з коректно працюючими процесами. У першому випадку групову розсилку вважають надійною, якщо можна гарантувати отримання повідомлення всіма правильно працюючими членами групи. Складність у тому, що додатково до всіх обмежень, пов'язаних із організацією процесів у групі, необхідно з'ясувати, які процеси входили до групи до отримання повідомлення.

Якщо вважати, що узгоджено, хто є членом групи, то за умови, що процеси в ході сеансу зв'язку не відмовляють, не входять до групи і не залишають її, то надійна групова розсилка означатиме, що будь-яке

повідомлення доставляється всім наявним членам групи. У простому випадку між членами групи немає домовленості про доставку повідомлень усім членам групи в певному порядку, але іноді така умова необхідна.

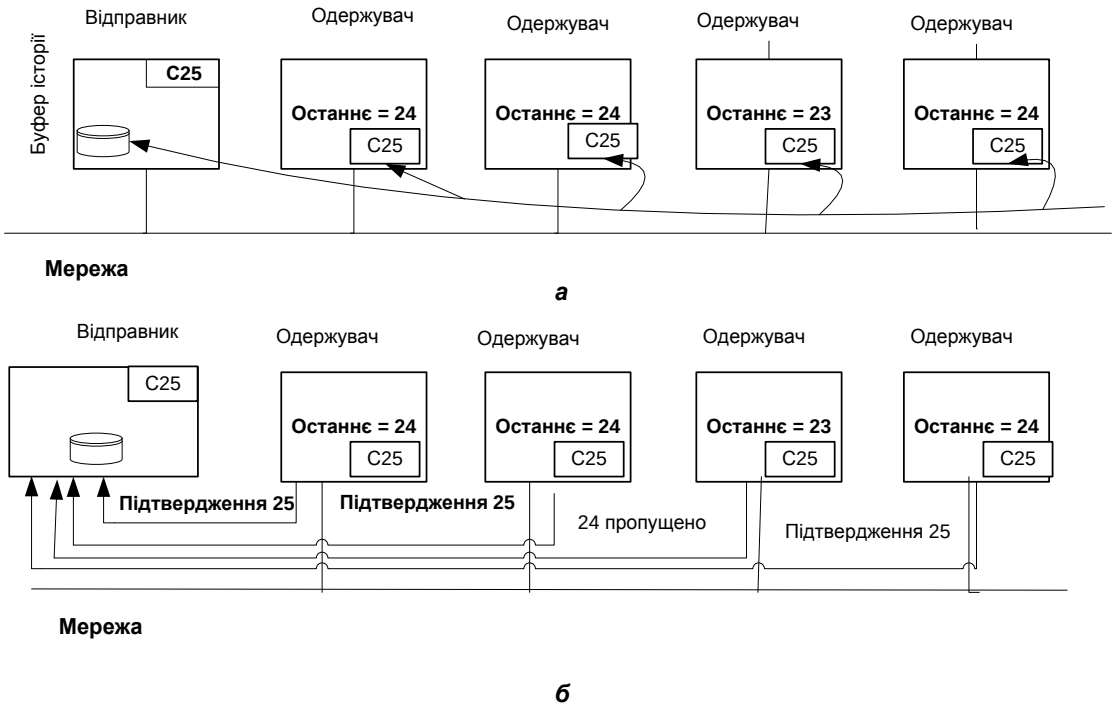


Рис. 2.6. Простий спосіб отримання надійної групової розсилки в тому разі, якщо передбачено, що одержувачі відомі і безвідмовні: передача повідомлень (*а*); отримання підтверджень прийому (*б*)

Така найбільш слабка форма надійної групової розсилки, яка порівняно просто реалізовується, застосовується в тому разі, якщо кількість одержувачів мала. Розглянемо випадок, коли один передавач має здійснити групову розсилку для декількох одержувачів. Припустімо, що базова система зв'язку забезпечує тільки ненадійну групову розсилку, тобто групові повідомлення можуть бути частково втрачені, а частково доставлені, але не всім потенційним одержувачам.

Простий спосіб реалізації надійної групової розсилки в таких умовах ілюструє рис. 2.6. Передавальий процес надає кожному повідомленню, що розсилається, послідовний номер. Припустімо, повідомлення приймаються в тому самому порядку, в якому

розсилаються, тоді одержувач легко виявить зникнення повідомлення. Вважаючи, що одержувачі знають, хто надіслав повідомлення, відправник просто зберігає повідомлення в буфері (буфері історії) до отримання підтверджень його прийому від усіх одержувачів. Якщо одержувач виявить зникнення повідомлення, то він повертає відправнику негативне підтвердження, запрошуючи повторну передачу, тоді як відправник, що не одержав підтвердження протягом заданого терміну, може виконати повторну передачу автоматично.

Так, щоб зменшити кількість повідомлень, що повертаються відправнику, підтвердження можуть вкладати в інші повідомлення. Крім того, повторна передача може здійснюватися як крізним каналом з кожним із процесів, які її потребували, так і за рахунок групової розсилки одного повідомлення всім процесам.

2.4.2. Масштабованість надійної групової розсилки

Основним недоліком описаної схеми групової розсилки є те, що вона не в змозі працювати з великою кількістю одержувачів. За наявності N одержувачів відправник повинен бути готовий обробляти як мінімум N підтверджень. Якщо одержувачів буде багато, відправник може бути просто «похованим» під відповідями. Цей ефект називають зворотним ударом (*feedback implosion*). Крім того, необхідно брати до уваги і той факт, що одержувачі можуть бути розкидані по всій глобальній мережі.

Одне з рішень проблеми полягає в тому, щоб заборонити одержувачам підтверджувати прийом повідомлення, натомість одержувач повинен посилати повідомлення відправнику тільки у разі втрати повідомлення. Якщо повертати лише негативні підтвердження, то негативний вплив ефекту зворотнього удару буде значно знижено,

але гарантії, що зворотний удар ніколи не відбудеться, як і раніше немає.

Іншим недоліком у разі отримання тільки негативних підтверджень є те, що відправник теоретично може бути вимушений постійно зберігати повідомлення в буфері історії. Оскільки відправник не в змозі дізнатися, чи було повідомлення доставлено всім одержувачам, він повинен завжди бути готовий до того, що один з одержувачів захоче повторно надіслати йому яке-небудь старе повідомлення. Відправник повинен видаляти повідомлення зі свого буфера історії після закінчення певного терміну, що збереже буфер від переповнення. Проте видалення повідомлення завжди загрожує тим, що один із запитів на повторну передачу не буде задоволений.

Наявні певні вимоги до надійних систем групової розсилки, які масштабуються.

Неієрархічне керування зворотним зв'язком. Ключове завдання під час створення масштабованих рішень для надійної групової розсилки – зменшення кількості відгуків, одержуваних відправником від одержувачів. У деяких глобальних інформаційних системах використовується популярна модель придушення відгуків (*feedback suppression*), яка лежить в основі протоколу надійної масштабованої групової розсилки (*Scalable Reliable Multicasting, SRM*), який працює таким чином**.**

Передусім у SRM одержувач ніколи не підтверджує успішного прийому повідомлення, надсилаючи відгук тільки у разі втрати повідомлення. Відстеження втрати повідомлення покладають на прикладну програму. Отже, як відгуки повертаються лише негативні підтвердження. Коли одержувач виявляє, що він втратив повідомлення, він, використовуючи групову розсилку, відправляє свій відгук решті членів групи, який змушує інших членів групи відмовитися від надсилання своїх власних повідомлень.

Приклад. Припустімо, що повідомлення *m* не дійшло до декількох членів групи. Кожний з них повинен надіслати відправнику *S* негативне підтвердження, щоб той повторно надіслав повідомлення *m*. Проте якщо повторне відправлення завжди виконується засобами групової розсилки, то буде цілком достатньо, якщо *S* одержить один-єдиний запит.

З цієї причини одержувач *R*, що не одержав повідомлення *m*, планує надіслати відгук з деякою випадковою затримкою, тобто запит на повторне надсилання відправляється тільки після деякого випадкового інтервалу часу. Проте, якщо в цей час до *R* надходить інший запит на повторне надсилання, *R* відмінить надсилання власного відгуку, знаючи, що повідомлення *m* скоро і так буде надіслане повторно. Таким чином, в ідеалі до *S* дійде тільки одне повідомлення, яке, у свою чергу, зумовить повторне надсилання *m*. Описану схему ілюструє рис. 2.7.

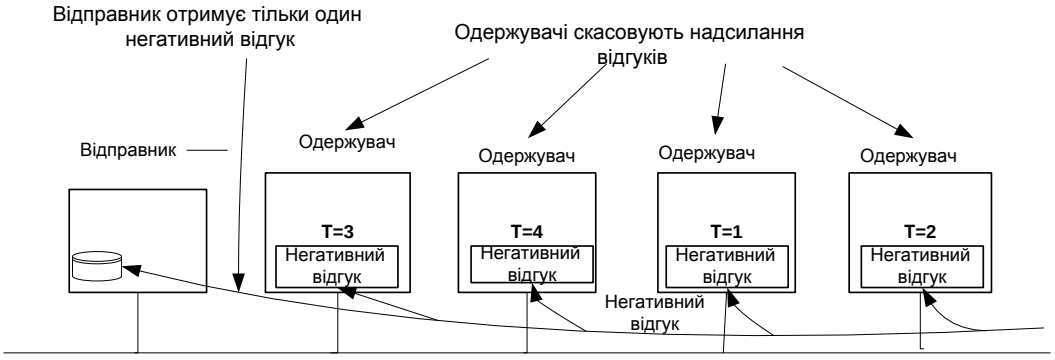


Рис. 2.7. Надсилання запитів на повторну передачу планують декілька одержувачів, але перший запит спричиняє відміну інших

Стримування відгуку приводить до значного підвищення масштабованості й використовується як базовий механізм для багатьох кооперативних Інтернет-прикладних програм, зокрема спільно використовуваних дошок оголошень. Проте, такий підхід має недоліки. Передусім, щоб гарантувати, що відправник одержить тільки один запит на повторне надсилання, необхідно дуже точно планувати відгук кожного з одержувачів, інакше багато одержувачів можуть надіслати свої відгуки одночасно, а тоді необхідно встановити таймери, за допомогою яких можна керувати процесом відправлення відгуків, але

урозкиданих по глобальній мережі групах процесів це є непростим завданням.

Групова розсилка відгуку перериває роботу і тих процесів, які успішно одержали повідомлення, тобто одержувачі вимушені приймати й обробляти непотрібне їм повідомлення. Єдине рішення цієї проблеми – виділити одержувачів, що залишилися без повідомлення *m*, в окрему групу для групової розсилки. На жаль, це рішення вимагає дуже ефективного керування групами, що у глобальних системах важко здійснити, тому найкращим підходом було б об'єднати одержувачів, що часто пропускають однакові повідомлення, у групу і використовувати один канал групової розсилки як для відгуків, так і для повторного надсилання повідомлень.

Для підвищення масштабованості SRM запропоновано, щоб одержувачі здійснювали локальне відновлення. Так, якщо одержувач, який успішно одержав повідомлення *m*, одержує потім запит на повторне надсилання, він сам ще до того, як запит дійде до дійсного відправника, може вирішити розіслати це повідомлення *m*.

Ієрархічне керування зворотним зв'язком. Щоб виконати вимогу масштабованості в дуже великих групах одержувачів, застосовують ієрархічний підхід до керування зворотним зв'язком у разі групової розсилки, основи якого ілюструє рис. 2.8.

Для простоти вважатимемо, що розсилає повідомлення великій групі одержувачів тільки один відправник, групу одержувачів розбито на дуже багато підгруп, організованих у вигляді дерева. Підгрупа, що містить відправника, становить корінь дерева. Всередині кожної з підгруп може використовуватися будь-яка схема групової розсилки, яка відповідає малій групі.

У кожній підгрупі визначається локальний координатор, який відповідає за обробку запитів на повторну передачу, які відправляють одержувачі, що входять у цю підгрупу. Локальний координатор

підтримує для цієї мети власний буфер історії. Якщо сам координатор пропускає повідомлення *m*, то він запрошує повторне надсилання цього повідомлення у координатора батьківської підгрупи. У схемі з підтвердженнями локальний координатор, отримавши повідомлення, надсилає підтвердження своєму предкові. Якщо координатор одержав підтвердження прийому повідомлення *m* від усіх членів своєї підгрупи, а також від усіх їхніх нащадків, то він може видалити *m* зі свого буфера історії.

Основна складність в ієрархічному підході – це побудова дерева, яке часто має будуватися динамічно. Одним із способів динамічної побудови дерева є використання дерева групової розсилки базової мережі, якщо воно існує. Для цього достатньо розширити завдання маршрутизаторів групової розсилки мережного рівня так, щоб вони могли виконувати функції локальних координаторів. На жаль, адаптувати таким чином наявну комп'ютерну мережу нелегко.

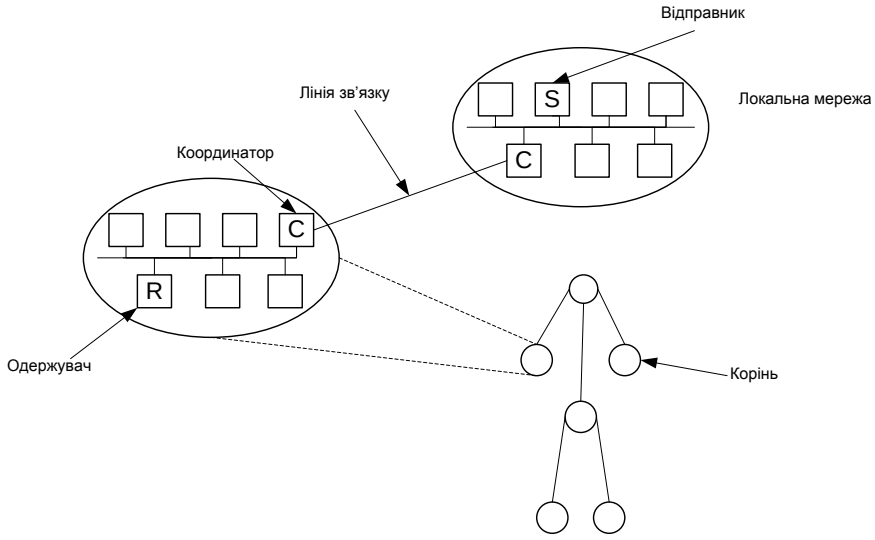


Рис. 2.8. Суть ієрархічної надійної групової розсилки: кожен локальний координатор пересилає повідомлення своїм нащадкам, а потім обробляє запити на повторне надсилання

2.4.3. Атомарна групова розсилка

Передусім, за моделі атомарної групової розсилки (*atomic multicast problem*) всі запити приходять до всіх серверів в однаковому порядку. Цю модель використовують у разі, коли потрібна надійна групова розсилка за наявності помилок в процесах. Часто в розподілених системах необхідно гарантувати доставку повідомлення або всім процесам в системі, або жодному з них. Окрім того, наявна вимога, щоб повідомлення доставлялися всім процесам у певному порядку.

Для розуміння причин важливості атомарності розглянемо реплікаційну базу даних, розроблену у вигляді надбудови над розподіленою системою, яка підтримує механізми надійної групової розсилки, зокрема вона дозволяє створювати групи процесів, яким можна гарантовано надсилати повідомлення. Реплікаційна база даних також побудована у вигляді групи процесів, по одному на репліку. Операція зміни завжди пересилається за допомого групової розсилки всім реплікам, а потім виконується локально, тобто використовується протокол активної реплікації.

Приклад. Нехай проводиться серія змін і в ході виконання однієї з них трапляється поломка репліки. Відповідно, оновлення цієї репліки не відбувається, тоді як оновлення інших реплік відбуваються успішно.

Природно, репліка відновлюється в тому вигляді, в якому вона була перед поломкою, тобто в ній не будуть враховані оновлення, що відбулися після поломки, тому важливо узгодити її з рештою реплік, що потребує повної інформації про те, які операції вона пропустила і в якому порядку вони виконувалися.

Якщо базова розподілена система підтримує атомарну групову розсилку, то операція зміни, розіслана всім реплікам перед тим, як відбулася поломка однієї з них, буде виконана на всіх коректно працюючих репліках або на жодній з них. У разі атомарної групової розсилки операція може бути виконана всіма робочими репліками, тільки якщо вони укладуть нову угоду про членство. Інакше кажучи,

зміна даних відбудеться тільки за умови, що решта реплік погодяться не вважати надалі зіпсовану репліку членом групи.

Коли зіпсована репліка буде відновлена, вона знову ввійде до групи, тобто не одержуватиме операцій зміни, поки знову не зареєструється як член групи. Вхід у групу вимагає, щоб її стан було узгоджено зі станом решти членів групи.

Відповідно, атомарна групова розсилка гарантує, що коректно працюючі процеси зберігають базу даних несуперечливою і здатні надати такого ж стану відновленій репліці, що знов увійшла до групи.

Віртуальна синхронність. Надійна групова розсилка за наявності помилок у процесах може бути виконана за допомогою механізму з розділенням засобів отримання і доставки повідомлення, який використовує групи процесів і зміни членства в групі. Схему роботи цього механізму подано на рис. 2.9. Повідомлення надсилаються і приймаються на комунікаційному рівні розподіленої системи. Прийняте повідомлення поміщається в локальний буфер комунікаційного рівня, а потім доставляється прикладній програмі, яка перебуває на більш високому рівні моделі OSI.

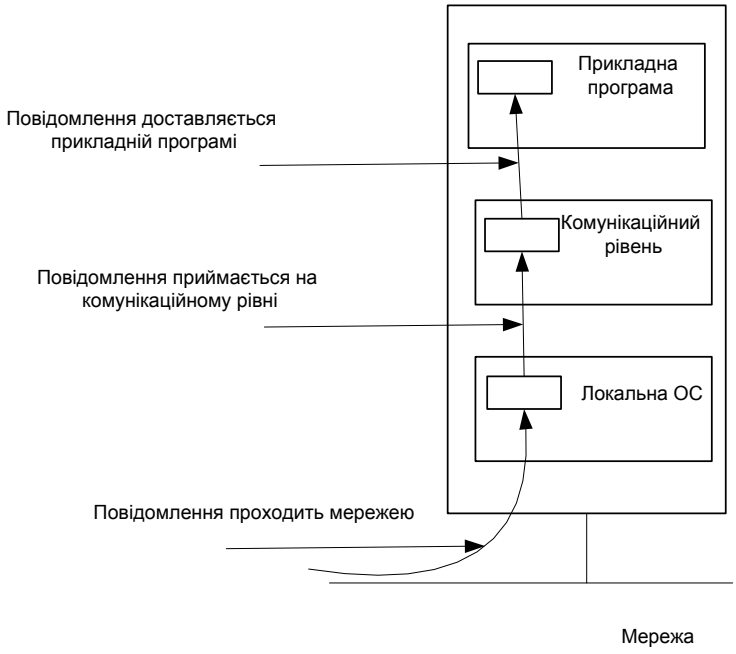


Рис. 2.9. Логічна організація розподіленої системи з розділенням засобів отримання і доставки повідомлення

Основна ідея атомарної групової розсилки полягає в тому, що розсилка членам групи повідомлення *m* однозначно асоціюється зі списком процесів-одержувачів. Список розсилки відповідає уявленню групи (*group view*), а точніше, уявленню набору процесів, що становлять групу, з якою працює відправник у момент розсилки повідомлення *m*. Важливо відзначити, що всі процеси у цьому списку мають однакове уявлення, тобто всі вони згодні з тим, щоб повідомлення *m* було доставлено кожному з них та більше нікому.

Припустімо, *m* розсилається в той момент, коли відправник має уявлення групи *G*. Окрім того, під час розсилки якийсь процес входить до групи або залишає її. Ця зміна складу групи оголошується всім процесам, що входять у *G*. Інакше кажучи, відбувається зміна уявлення (*view change*) за рахунок групової розсилки повідомлення *vc* про вхід процесу в групу або вихід з неї, тобто наявні два розісланих повідомлення, які перебувають «в дорозі» одночасно: *m* і *vc*. Необхідно гарантувати, що *m* або буде доставлено всім процесам у *G* до того, як будь-який з них одержить повідомлення *vc*, або взагалі не буде доставлено.

Є тільки один випадок, коли доставка *m* допускає відмову: коли склад групи змінюється в результаті поломки відправника повідомлення *m*. У цьому разі або всі члени *G* повинні одержати повідомлення про аварійне завершення процесу додавання нового елемента або жоден з них. Проте повідомлення *m* може бути просто проігноровано всіма членами групи, що відповідатиме ситуації поломки відправника до початку розсилки *m*.

Ця найбільш жорстка форма надійної групової розсилки гарантує, що розіслане повідомлення відповідно до уявлення групи *G*, буде доставлено кожному нормально функціонуючому процесу, що входить у *G*. Якщо відправник повідомлення у процесі розсилки відмовить, то повідомлення буде або прийнято, або проігноровано рештою процесів.

Надійну групову розсилку з такими властивостями називають віртуально синхронною (*virtually synchronous*).

Розглянемо чотири процеси. У деякий момент часу процес *P1* вступає у групу, яка міститиме процеси *P1*, *P2*, *P3* і *P4*. Після розсилки декількох повідомлень процес *P3* відмовляє. До поломки він встигає відправити чергове повідомлення процесам *P2* і *P4*, але не *P1*. У такому разі віртуальна синхронність гарантує, що повідомлення взагалі не буде доставлено, тому що імітується стан, неначебто повідомлення перед поломкою *P3* взагалі не відправлялося.

Після того як процес *P3* буде видалений з групи, взаємодія між членами, що залишилися у групі, продовжиться. Пізніше, коли процес *P3* буде відновлений, його можна буде знову залучити у групу, а потім актуалізувати його стан.

Принцип віртуальної синхронності полягає в тому, що всі групові розсилки відбуваються у проміжках між змінами уявлення. Інакше кажучи, зміни уявлень є бар'єрами, через які розсилки не в змозі пройти. У цьому сенсі цей принцип аналогічний принципу використання змінних синхронізації в розподілених сховищах даних. Перш ніж зміна уявлень дозволеною для використання, мають завершитися всі групові розсилки, які виконувались під час зміни уявлень.

Впорядкування повідомлень. Віртуальна синхронність дозволяє розробнику прикладних програм уважати, що групові розсилки відбуваються в різні епохи (проміжки часу), відокремлені одна від одної змінами у складі групи. Загалом можна виокремити чотири варіанти групових розсилок з різним порядком проходження: невпорядковані; у порядку FIFO; причинно впорядковані та повністю впорядковані групові розсилки.

Надійна невпорядкована групова розсилка (*reliable unordered multicast*) – це віртуально синхронна групова розсилка, що ніяк не

гарантує порядку надходження повідомлень до різних процесів. Щоб зрозуміти це визначення, розглянемо надійну групову розсилку, реалізовану за допомогою бібліотеки, в якій є примітиви відправлення й отримання. Операція відправки блокує процес, який викликав її до передачі йому повідомлення.

У разі надійної групової розсилки в порядку FIFO (*reliable* FIFO–*ordered multicast*) комунікаційний рівень має доставляти вхідні повідомлення кожному з процесів у тому порядку, в якому вони були відправлені. Цього правила дотримуються всі процеси у групі, але обмеження на доставку повідомлень, які надіслано різними відпраниками, відсутні.

У разі надійної причинно впорядкованої групової розсилки (*reliable causally-ordered multicast*) повідомлення доставляються у порядку потенційного причинного зв'язку між ними, тобто, якщо повідомлення *m1* через певну умову передує повідомленню *m2*, незалежно від того, надсилалися вони одним відправником або кількома, комунікаційний рівень кожного з одержувачів завжди доставлятиме *m2* після того, як прийме і доставить *m1*.

Окрім розглянутих варіантів упорядкування, можна ввести додаткові обмеження на процес упорядкування повідомлень у разі групової розсилки, які визначатимуть повну впорядкованість повідомлень.

Повністю впорядкована групова розсилка (*total-ordered multicast*) означає, що незалежно від того, як саме впорядкована доставка повідомлень (причинно, по FIFO або не впорядкована зовсім), додатково потрібно, щоб повідомлення доставлялися всім членам групи в однаковому порядку.

Віртуальну синхронну надійну групову розсилку, яка підтримує повністю впорядковану доставку називають *атомарною груповою розсилкою* (*atomic multicasting*).

Реалізація віртуальної синхронності. Розглянемо реалізацію віртуально синхронної надійної групової розсилки. Прикладом такої реалізації є відмовостійка розподілена система Isis, яку декілька років використовували на практиці.

Надійна групова розсилка 128е блокувал на використанні наявних у базовій мережі, зокрема в мережі TCP, механізмів надійного крізного зв'язку. Розсилка повідомлення *m* групі процесів реалізована за рахунок надійного відправлення повідомлення *m* кожному члену групи. Таким чином, хоча будь-яка передача гарантовано успішна, немає ніяких гарантій, що повідомлення *m* буде одержано всіма членами групи. Зокрема, відмова відправника може відбутися до того, як він передасть усім членам групи повідомлення *m*.

Окрім надійної крізної комунікації, передбачено, що повідомлення з одного джерела надсилаються засобами комунікаційного рівня у тому самому порядку, в якому джерело їх відправляє. На практиці ця вимога вирішується за рахунок використання для крізної комунікації з'єднань TCP.

Основним завданням у разі використання надійної групової розсилки є необхідність гарантувати, щоб усі повідомлення, надіслані уявленню *G*, доставлені всім правильно працюючим процесам з *G* до чергової зміни складу групи. Для цього передусім потрібно, щоб кожний процес з уявлення *G* отримав усі відправлені туди повідомлення. Відзначимо, що оскільки відправник повідомлення *m* в уявлення *G* може відмовити до завершення розсилки, деякі процеси в *G* можуть ніколи не отримати *m*. Оскільки відправник відмовив, ці процеси мають одержати *m* від іншого відправника.

Рішення цієї проблеми полягає в тому, щоб кожен процес у *G* зберігав *m* доти, поки він точно не знатиме, що це повідомлення одержали всі члени *G*. Якщо повідомлення *m* одержали всі члени уявлення *G*, то *m* називають стійким (*stable*) повідомленням.

Доставляти дозволено тільки стійкі повідомлення. Щоб гарантувати стійкість, достатньо вибрати G випадковий робочий процес і вимагати від нього відправлення повідомлення m решті процесів.

Припустімо, що поточне уявлення – це G_i, але є необхідність установити нове уявлення G_{i+1}, яке відрізняється від G_i максимум одним процесом. Процес P дізнається про зміну уявлення, одержуючи відповідне повідомлення, яке може надійти від процесу, який хоче ввійти до групи або залишити її, або від процесу, що виявив відмову одного з процесів, що входять у G. Збійний процес після цього повинен бути видалений, як показано на рис. 2.10, *а*.

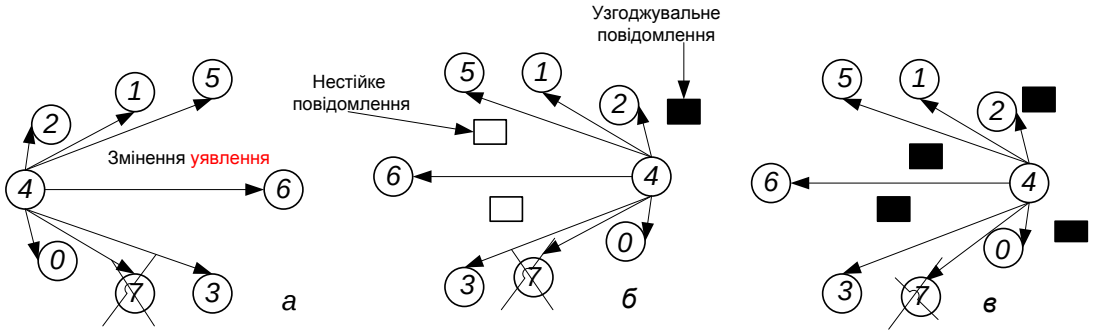


Рис. 2.10. Процес *4* сповіщає, що процес *7* відмовив, розсилаючи повідомлення про зміну уявлення (*а*); процес *6* розсилає всі свої нестійкі повідомлення разом з узгоджувальними повідомленнями (*б*); процес *6* установлює нове уявлення, одержавши узгоджувальне повідомлення від іншого відправника (*в*)

Коли процес P одержує повідомлення про зміну уявлення на G_{i+1}, він в передусім пересилає в G копії всіх нестійких повідомлень, щоб вони дійшли до кожного процесу, що входить у G, після чого позначає їх як стійкі. У разі використання надійної групової розсилки крізні взаємодії передбачаються надійними, так що переслані повідомлення ніколи не втрачаються, тобто всі повідомлення, відправлені до G і одержані хоч би одним процесом, будуть одержані всіма правильно працюючими процесами в G. Відзначимо, що для пересилання нестійких повідомлень достатньо вибрати одного координатора.

Щоб позначити, що процес *P* більше не має нестійких повідомлень і готовий установити уявлення G_{i+1} після того, як це зможуть здійснити інші процеси, він розсилає в уявлення G_{i+1} узгоджувальне повідомлення (*flush message*), як показано на рис. 2.10, *б*. Після того як процес *P* прийме узгоджувальне повідомлення для уявлення G_{i+1} від якого-небудь іншого процесу, він може встановити нове уявлення, як показано на рис. 2.10, *в*.

Якщо інший процес *Q* уважатиме, що поточне уявлення – G_i, то, одержавши повідомлення *m*, надіслане відповідно до G_i, він доставить його, враховуючи всі додаткові обмеження щодо порядку повідомлень. Якщо він уже приймав *m*, то визнає це повідомлення повторним і пропустить його.

Оскільки процес *Q* врешті-решт прийме повідомлення про зміну уявлення на G_{i+1}, він також спочатку перешле всі свої нестійкі повідомлення, після чого оголосить про закінчення цієї роботи відправленням повідомлення членам узгоджувального уявлення G_{i+1}. Відзначимо, що згідно з порядком повідомлень, установленим базовим комунікаційним рівнем, узгоджувальне повідомлення завжди приймається після нестійкого повідомлення, надісланого тим самим процесом.

Основний недолік описаного протоколу полягає у тому, що він не запобігає збоям процесів у момент оголошення нової зміни уявлення, тому передбачено, що доти, поки нове уявлення G_{i+1} не буде встановлено кожним членом уявлення G_{i+1}, жоден процес із G_{i+1} не відмовить (відмова процесу зумовить виникнення нового уявлення G_{i+2}). Ця проблема розв’язується за рахунок оголошення про зміну уявлень для будь-якого уявлення G_{i+k} ще до того, як попередні зміни будуть встановлені всіма процесами.

2.5. Розподілене підтвердження

Завдання атомарної групової розсилки є прикладом більш загального, відомого під назвою *«розподілене підтвердження»* (*distributed commit*), яке містить операції, що відбуваються з кожним членом групи процесів або з жодним з них. У разі надійної групової розсилки операцією буде доставка повідомлення, а у разі розподілених транзакцій – підтвердження транзакції на одному із сайтів, задіяних у транзакції.

Розподілене підтвердження часто організовується за допомогою координатора. У простій схемі координатор повідомляє всі інші процеси, які також беруть участь у роботі (учасники), чи можуть вони локально здійснити запрошувану операцію. Цю схему називають *протоколом однофазного підтвердження* (*one-phase commit protocol*), який має один суттєвий недолік. Якщо один з учасників насправді не може здійснити операцію, то він не в змозі повідомити про це координатору. Так, у разі розподілених транзакцій локальне підтвердження може виявитися неможливим через те, що воно порушуватиме обмеження керування паралельним виконанням.

Для практичного застосування зазвичай використовують *протокол двофазного підтвердження*, основний недолік якого полягає в тому, що він не може ефективно обробляти помилки координатора, тому було розроблено *протокол трифазного підтвердження*.

2.5.1. Двофазне підтвердження

Розглянемо розподілену транзакцію, яка передбачає участь багатьох процесів, кожний з яких працює на окремій машині. Якщо помилок немає, то протокол двофазного підтвердження 2PC (*Two-phase*

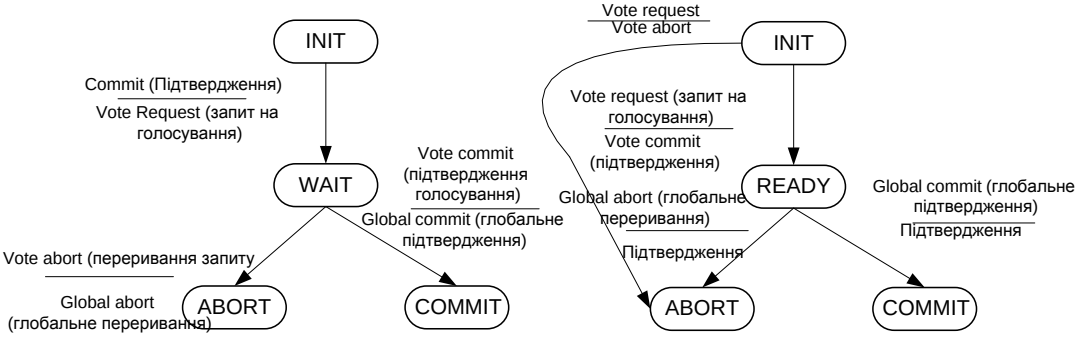
Commit Protocol) складається з двох фаз, кожна з яких містить два кроки.

Координатор розсилає всім учасникам повідомлення *VOTE_REQUEST*. Після того, як учасник одержить повідомлення *VOTE_REQUEST*, він повертає координатору або повідомлення *VOTE_COMMIT*, указуючи, що він готовий локально підтвердити свою частину транзакції, або повідомлення *VOTE_ABORT*, щоб перервати виконання цієї частини транзакції.

Координатор збирає відповіді учасників. Якщо всі учасники проголосували за підтвердження транзакції, то координатор надсилає всім учасникам повідомлення про глобальне підтвердження *GLOBAL_COMMIT*. Якщо ж хоч один учасник проголосував за переривання транзакції, то координатор ухвалює відповідне рішення і розсилає повідомлення про глобальне переривання процесу виконання транзакції *GLOBAL_ABORT*.

Кожний з учасників, що проголосували за підтвердження, чекає підсумкового рішення координатора. Одержуючи повідомлення *GLOBAL_COMMIT*, учасник локально підтверджує транзакцію. У разі ж отримання повідомлення *GLOBAL_ABORT* транзакція локально переривається.

Перша фаза (голосування) складається з кроків 1 і 2, друга (рішення) – з кроків 3 і 4, які показано на діаграмі кінцевого автомата (рис. 2.11).



а *б*

Рис. 2.11. Координатор у протоколі 2PC у вигляді кінцевого автомата (*а*); учасник у вигляді кінцевого автомата (*б*)

Коли під час використання базового протоколу 2PC у системі з’являються помилки, виникають блокування. Передусім відзначимо, що як координатор, так і учасники мають такі стани, в яких вони блокуються і чекають надходження повідомлень. Відповідно, якщо процес відмовляє, то робота протоколу легко може бути порушена, оскільки інші процеси надто довго чекатимуть від нього повідомлення. З цієї причини вводяться механізми тайм-ауту.

Наявні всього три стани, в яких координатор або учасник можуть, будучи заблокованими, чекати надходження повідомлення: *INIT*, *WAIT* та *READY*.

Учасник може, перебуваючи в початковому стані *INIT*, чекати повідомлення *VOTE_REQUEST* від координатора. Якщо протягом деякого часу цього повідомлення немає, то він робить висновок про необхідність локального переривання транзакції та надсилає координатору повідомлення *VOTE_ABORT*.

Так само координатор, блокований у стані *WAIT*, чекає надходження голосів усіх учасників. Якщо протягом деякого часу не будуть зібрані всі голоси, то координатор вирішує, що транзакцію необхідно перервати і розсилає всім учасникам повідомлення *GLOBAL_ABORT*.

І, нарешті, учасник може бути блокований у стані *READY*, чекаючи глобального оголошення результатів голосування, які розсилає координатор. Якщо це повідомлення за певний період часу не надходить, то учасник не може просто перервати транзакцію, натомість він повинен визначити: яке саме повідомлення надсилав координатор. Найпростішим рішенням цього питання буде блокування учасника до моменту відновлення координатора.

Правильнішим рішенням буде дозволити учаснику *P* контакт з іншим учасником *Q*, щоб він міг за поточним станом *Q* вирішити, що ж йому робити далі. Нехай *Q* набув стану *COMMIT*. Це можливо лише за

умови, що координатор до поломки надіслав *Q* повідомлення *GLOBAL_COMMIT*. До *P* це повідомлення не дійде. Відповідно, *P* тепер може дійти висновку про необхідність локального підтвердження. Так само якщо *Q* перебуває у стані *ABORT*, *P* може впевнено переривати свою частину транзакції.

Тепер припустимо, що *Q* перебуває у стані *INIT*, що можливо, якщо координатор розішле всім учасникам повідомлення *VOTE_REQUEST*, і це повідомлення дійде до *P* (який відповість на нього відправленням повідомлення *VOTE_COMMIT*), але не дійде до *Q*. Інакше кажучи, координатор може відмовити під час розсилання повідомлення *VOTE_REQUEST*, тоді правильно буде перервати транзакцію: і *P*, і *Q* можуть здійснити перехід у стан *ABORT*.

Найскладнішою є ситуація, коли як *Q* перебуватиме у стані *READY*, чекаючи відповіді від координатора, так і всі учасники перебувають у стані *READY*, конкретного рішення за таких умов прийняти неможливо. У зв’язку з тим, що всі учасники готові підтвердити транзакцію, але не мають підтвердження координатора, то протокол блокується до моменту відновлення координатора.

Щоб гарантувати, що процес дійсно відновився, потрібно, щоб він зберігав свій стан за допомогою засобів тривалого зберігання даних. Наприклад, якщо учасник перебував у стані *INIT*, то після відновлення він може вирішити локально перервати транзакцію і повідомити про це координатору. Так само якщо він уже ухвалив рішення і у момент поломки перебував у стані *COMMIT* або *ABORT*, то у разі відновлення він знову набуде вибраного стану і повторно повідомить своє рішення координатору.

У разі коли учасник відмовляє, перебуваючи у стані *READY*, то під час відновлення він не може визначитись на підставі власної інформації, що робити далі: завершувати або переривати транзакцію. Відповідно, щоб з’ясувати це, цей процес має зв’язатися з іншими

учасниками, як це відбувається у разі закінчення тайм-ауту в стані *READY*.

Координатор має тільки два критичні стани, які треба контролювати. Починаючи протокол 2PC, він повинен зберегти відомості про те, що він перебуває в початковому стані *WAIT*, щоб мати можливість, у разі потреби, після відновлення повторно розіслати повідомлення *VOTE_REQUEST* усім учасникам. Окрім того, якщо у другій фазі він ухвалив рішення, то слід зберегти це рішення, щоб після відновлення координатора його також можна було розіслати повторно.

Схема дій координатора така: координатор починає з групової розсилки всім учасникам повідомлення *VOTE_REQUEST* для того, щоб зібрати їх голоси. Він записує цю дію, після чого набуває стану *WAIT*, у якому чекає надходження голосів учасників.

Якщо зібрано не всі голоси, а протягом заданого часу не надіслано тих, яких не вистачає, координатор робить висновок, що один або більше учасників перебувають у неробочому стані. Це означає, що він має перервати транзакцію і розіслати решті учасників повідомлення *GLOBAL_ABORT*.

Якщо не відбулося ніяких відмов, то координатор збере всі голоси. Якщо всі учасники, так само як і координатор, проголосували за підтвердження, то повідомлення *GLOBAL_COMMIT* буде спочатку записано в журнал, а потім розіслано всім процесам. Інакше координатор розішле повідомлення *GLOBAL_ABORT* (заздалегідь також записавши його в локальний журнал).

На початку обробки певної елементарної неподільної розподіленої транзакції процес чекає запиту на голосування від координатора. Відзначимо, що це очікування виконується в окремому потоці виконання, що здійснюється в адресному просторі процесу. Якщо повідомлення не надходить, то транзакція переривається, ймовірно, через відмову координатора.

Проте якщо учасник чекає рішення координатора весь відведений на це термін і не одержує його, то він виконує протокол припинення очікування, спочатку розсилаючи решті процесів повідомлення *DECISION_REQUEST*, а потім блокуючись в очікуванні відповіді. Коли надходить відповідь (можливо, від координатора, який уже встиг відновитися), учасник записує рішення в локальний журнал і продовжує свою роботу у відповідності з отриманним рішенням.

Кожен учасник має бути готовий до того, що інший учасник запитає у нього глобальне рішення. Для цього кожний з учасників запускає окремий потік виконання паралельно зі своїм основним, який блокується до отримання запиту про рішення. Він може відповісти іншим процесам, тільки якщо асоційований з ним учасник уже прийняв остаточне рішення. Інакше кажучи, якщо в локальний журнал було записано повідомлення *GLOBAL_COMMIT* або *GLOBAL_ABORT*, то можна бути впевненим, що координатор надіслав своє рішення усім процесам. Крім того, якщо асоційований з потоком учасник перебуває у стані *INIT*, то він може вирішити надіслати повідомлення *GLOBAL_ABORT*. У всіх інших станах цей потік не зможе відповісти процесу, який надіслав запит, і він залишиться без відповіді.

Можлива ситуація, коли учаснику доведеться встановити блокування до відновлення координатора, якщо всі учасники одержать і оброблять повідомлення координатора *VOTE_REQUEST* і в цей час координатор зламається. У цьому разі учасники не зможуть спільно вирішити, яку дію їм виконати. З цієї причини протокол 2PC також називають протоколом блокувального підтвердження (*blocking commit protocol*).

Розрізняють декілька способів уникнути блокування:

1) використання примітиву групової розсилки, коли одержувач негайно розсилає у відповідь повідомлення решті процесів, такий

підхід дозволяє учаснику прийняти остаточне рішення навіть у тому разі, якщо координатор ще не відновився;

2) використання протоколу трифазного підтвердження.

2.5.2. Трифазне підтвердження

Недоліком протоколу двофазного підтвердження є те, що у разі непрацездатності координатора учасники не зможуть прийняти остаточне рішення, тому їм доводиться чекати відновлення координатора, перебуваючи у заблокованому стані. Було запропоновано варіант протоколу *2PC*, названий протоколом трифазного підтвердження (*Three-phase Commit Protocol, 3PC*), який запобігає блокуванню процесів у разі появи помилок через аварійну зупинку. Хоча протокол *3PC* часто згадується в літературних джерелах, на практиці його використовують рідко, оскільки рідко виникають умови, в яких блокується *2PC*. Як і *2PC, 3PC* формулюється в термінах координатора і набору учасників. Відповідні алгоритми їх роботи у вигляді кінцевих автоматів показано на рис. 2.12. Суть алгоритму роботи протоколу полягає в тому, що стани координатора і будь-якого з учасників задовольняють двом умовам;

- не має такого стану, з якого може бути здійснений прямий перехід як у стан *COMMIT*, так і у стан *ABORT*.
- не має такого стану, в якому неможливо прийняти остаточне рішення, але можливий перехід у стан *COMMIT*.

Ці дві умови необхідні й достатні для того, щоб протокол підтвердження був неблокувальним.

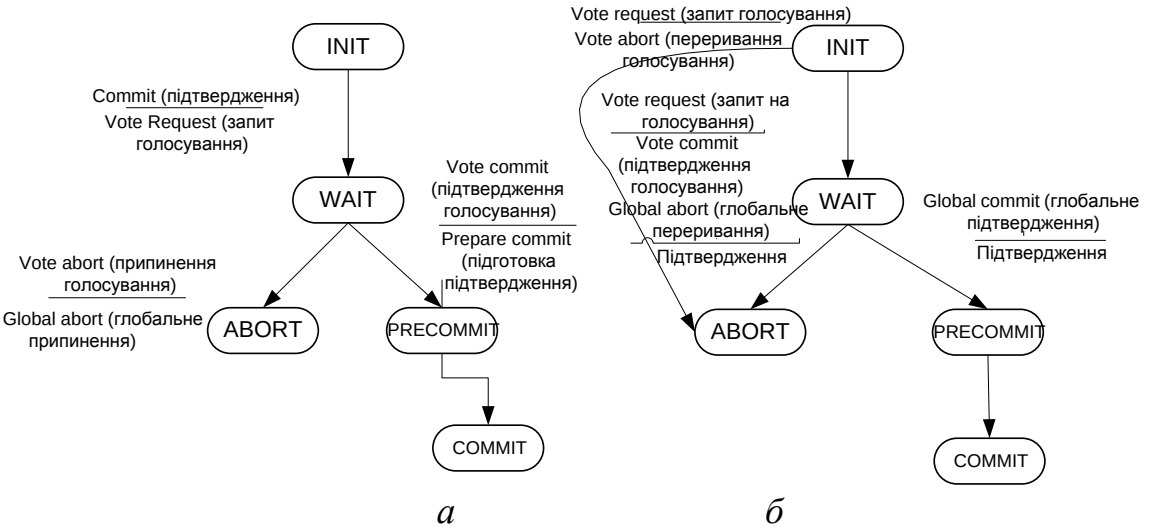


Рис. 2.12. Алгоритм роботи координатора у протоколі 3PC у вигляді кінцевого автомата (*а*). Учасник у вигляді кінцевого автомата (*б*)

2.6. Відновлення

2.6.1. Основні поняття. Стійкі сховища

Основа відмовостійкості – виправлення системи після помилок, причому помилкою вважається відмова частини системи. Основна ідея механізму виправлення помилок полягає в заміні помилкового стану безпомилковим. Розрізняють два основні способи відновлення після помилок: зворотне та пряме виправлення.

У разі зворотного виправлення (*backward recovery*) основне завдання полягає в поверненні системи з поточного помилкового до попереднього безпомилкового стану. Для цього необхідно час від часу записувати стан системи і відновлювати її у попередньому стані, якщо виникає збій. Під час кожного запису поточного стану системи (або його частини) створюється *контрольна точка* (*checkpoint*).

Другий варіант виправлення помилок – *пряме виправлення* (*forward recovery*), коли у разі входу системи в помилковий стан замість відкату до попередньої контрольної точки, робиться спроба перевести

систему в новий коректний стан, у якому вона могла б продовжувати працювати. Основною складністю механізмів прямого виправлення помилок є те, що для них необхідно можливі помилки знати наперед, і лише тоді вдасться усунути ці помилки і перейти в новий стан.

Відмінності між прямим і зворотним механізмами виправлення помилок легко зрозуміти на прикладі реалізації надійного зв'язку, за якого основним способом відновлення втраченого пакета є його повторне надсилання відправником. Повторна передача пакета означає повернення до попереднього правильного стану, а саме до стану, коли починав передаватися втрачений пакет. Отже, надійний зв'язок, організовуваний за допомогою повторного відправлення пакетів є прикладом зворотного виправлення помилок.

Альтернативний підхід полягає у використанні методу корекції зруйнованої інформації, коли втрачений пакет створюється з інших успішно доставлених пакетів. Так, у разі використання блочного кодування від втрат *(n, k)* набір з *k* початкових пакетів перетвориться в набір *n* декодованих пакетів, так що будь-який набір з *k* декодованих пакетів придатний для реконструкції *k* початкових пакетів, де типовими значеннями *n* і *k* є 16 - 32, причому $k < n \leq 2k$. Якщо доставлених пакетів недостатньо, то відправнику доведеться продовжувати передавати пакети доти, поки не вдасться відновити раніше втрачений пакет. Корекція зруйнованої інформації є типовим прикладом прямого виправлення помилки.

Для виправлення помилок у розподілених системах як основний механізм найчастіше використовують методи зворотного виправлення, важливою перевагою якого є те, що це загальний метод, який не залежить від конкретної системи або процесу. Інакше кажучи, він може бути інтегрований у розподілену систему, наприклад у її проміжний рівень у вигляді служби загального призначення.

Проте зворотне виправлення помилок також створює певні проблеми. По-перше, відновлення системи або процесу до попереднього стану є достатньо затратним з погляду продуктивності.

По-друге, оскільки механізми зворотного виправлення помилок не залежать від конкретної розподіленої прикладної програми, в якій вони використовуються, неможливо дати які-небудь гарантії того, що після виправлення помилок та сама або подібна помилка не виникне знов. Для запобігання повторної появи помилок їх обробка зазвичай виконується в циклі. Застосування механізмів зворотного виправлення помилок не дозволяє досягти повністю прозорого виправлення помилок.

Крім того, хоча під час зворотного виправлення помилок використовують контрольні точки, в деякі стани повернутися просто неможливо. Наприклад, неможливе відновлення до попереднього стану більшості UNIX-систем після введення команди, яка вказує шлях до певного ресурсу, і вибору неправильного каталогу.

Контрольні точки дозволяють повернутися до попереднього правильного стану. Проте створення контрольної точки часто спричиняє значні витрати, а отже, значні втрати продуктивності, тому багато відмовостійких розподілених систем поєднують створення контрольних точок з протоколюванням повідомлень (*message logging*). У цьому разі після створення контрольної точки процес записує свої повідомлення перед відсиланням до журналу, тобто здійснює **протоколювання відправником** (*sender-based logging*).

Альтернативне рішення – дозволити приймальному процесу протоколювати вхідні повідомлення перед їх доставкою прикладній програмі. Цю схему називають **протоколювання одержувачем** (*receiver-based logging*). Якщо у приймальному процесі виникає відмова, то він має відновити стан останньої збереженої контрольної точки, після чого може повторно запустити всі прийняті після неї

повідомлення. Таким чином, поєднання контрольних точок і протоколювання повідомлень дозволяє відновити значно пізніший стан, ніж стан, збережений останньою контрольною точкою, без додаткових витрат на створення контрольних точок.

Є ще одна важлива відмінність між контрольними точками і схемами, які ґрунтуються на застосуванні журналів. У системі, яка використовує тільки контрольні точки, на відміну від систем з журналізацією, процес може бути відновлений тільки зі стану контрольної точки. Але його поведінка в цьому стані може відрізнятися від поведінки перед помилкою. Наприклад, якщо періоди між сеансами задані не жорстко, повідомлення тепер можуть надходити в іншому порядку, зумовлюючи іншу реакцію одержувача, натомість у разі протоколювання повідомлень відбувається повторення всіх подій, що відбувалися з моменту останньої контрольної точки. Це повторення полегшить взаємодію системи із зовнішніми користувачами.

Розглянемо ситуацію, коли помилка спричинена неправільними даними, які вводить користувач. Якщо використовувати виключно контрольні точки, то єдине, що може система, – це повернутися до стану контрольної точки, яка передувала прийому введених користувачем даних. Протоколюючи повідомлення можна використовувати старішу контрольну точку, після чого, повторюючи події, вийти до точки, де користувач повинен вводити дані. На практиці поєднання невеликої кількості контрольних точок і протоколювання повідомлень використовується набагато частіше, ніж схеми з множиною контрольних точок.

Стійкі сховища. Щоб мати можливість відновити попередній стан системи, потрібно зберігати потрібну для відновлення інформацію в безпеці. Це означає, що збережена інформація має витримувати відмови системи і сайтів, а можливо, і різні пошкодження сховища.

Стійкі сховища відіграють важливу роль у відновленні розподілених систем.

Сховища поділяють на три категорії. 1) стандартна оперативна пам'ять, яка стирається у разі вимикання живлення або перезавантаження машини; 2) дискові сховища, які стійкі до помилок процесора, але не здатні вижити у разі поломки головок диска; 3) стійкі сховища (*stable storage*), розроблені для виживання в будь-яких ситуаціях, окрім катастроф, зокрема повеней або землетрусів.

2.6.2. Створення контрольних точок

У відмовостійких розподілених системах зворотне виправлення помилок вимагає, щоб система регулярно записувала свій стан у стійке сховище даних, зокрема, несуперечливий глобальний стан, або розподілений знімок стану (*distributed snapshot*). Якщо в розподіленому знімку стану процес *P* був збережений у момент отримання повідомлення, має також бути процес *Q*, збережений у момент передачі повідомлення, за якого, як виняток, повідомлення може надійти і зі сторони по відношенню до процесу *P*.

У схемах зворотного виправлення помилок кожен процес час від часу записує свій стан у локальне стійке сховище. Для відновлення після системної помилки необхідно з цих локальних станів відтворити несуперечливий загальний стан, для чого краще за все відновити останній розподілений знімок стану, або лінію відновлення (recovery line).

Незалежне створення контрольних точок. На жаль, розподілена природа контрольних точок, коли кожен процес просто час від часу записує свій локальний стан, ніяк не координуючи свої дії з іншими, може сильно утруднити пошуки межі відновлення. Для знаходження лінії відновлення необхідно, щоб кожний процес відкатався до

останнього записаного стану. Якщо ці локальні стани не утворюють розподіленого знімка стану, буде потрібно здійснити новий відкіт і т.д. Цей процес каскадного відкату може зумовити ефект доміно (*domino effect*), як показано на рис. 2.13.

Розглянемо ситуацію, яка показана на рис. 2.13, у разі помилки у процесі *P2* необхідно відновити його стан, яким його було записано в останній контрольній точці, процес *P1* також доведеться відкатити. Два останніх записаних локальних стани не утворюють глобального несуперечливого стану: стан, записаний у *P2*, вказує на прийом повідомлення *m*, але жоден процес не може бути названий його відправником. Відповідно, необхідно здійснити відкат *P2* до попереднього стану.

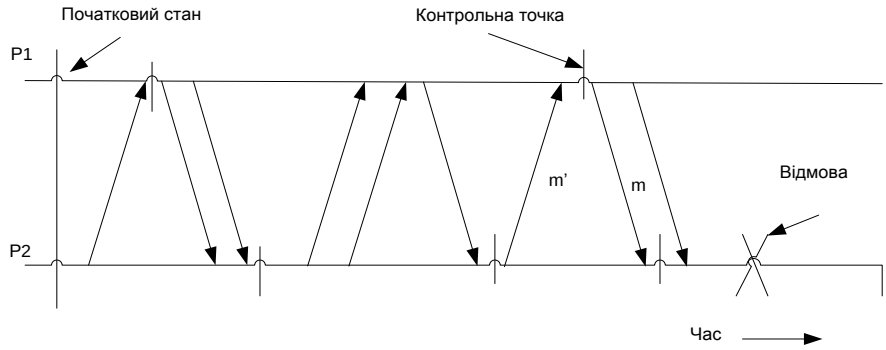


Рис. 2.13. Ефект доміно

Проте наступний стан, у якому буде відновлено процес *P2*, також не може використовуватися як частина розподіленого знімка стану. В цьому разі процес *P1* записано таким, що приймає повідомлення *m'*, а подія його відправлення знову не буде записана. Таким чином, з'явиться необхідність відкату процесу P1 до попереднього стану. Отже межею відновлення насправді є початковий стан системи.

Оскільки процеси створюють локальні контрольні точки незалежно один від одного, цей метод називають також методом *незалежного створення контрольних точок* (*independent checkpointing*), альтернативою якому є глобально координоване створення

контрольних точок. Проте координація вимагає глобальної синхронізації, яка може зменшити продуктивність розподіленої системи. Інші недоліки незалежного створення контрольних точок полягають у тому, що будь-яке локальне сховище потрібно час від часу чистити, наприклад, запускаючи спеціальну програму «збирання сміття» та найсуттєвішим недоліком є необхідність розрахунку межі відновлення.

Координоване створення контрольних точок. У разі координованого створення контрольних точок (*coordinated checkpointing*) усі процеси синхронізовані для того, щоб запис їх станів у локальні стійкі сховища відбувався одночасно. Основна перевага координованого створення контрольних точок полягає в тому, що записаний стан автоматично стає глобально несуперечливим, що дозволяє уникнути каскадного відкату, який зумовлює ефект доміно. Для координованого створення контрольних точок може використовуватися алгоритм розподіленого знімка стану, який є прикладом неблокувальної координації контрольних точок.

Найпростішим рішенням уважають використання двофазного протоколу блокування: спочатку координатор розсилає всім процесам повідомлення *CHECKPOINT_REQUEST*. Коли процес одержує це повідомлення, він створює локальну контрольну точку, починає вибудовувати в чергу всі наступні повідомлення, що надходять до нього від працюючого прикладного програмного забезпечення і відправляє координатору підтвердження створення контрольної точки. Після того, як координатор одержить підтвердження від усіх процесів, він розсилає повідомлення *CHECKPOINT_DONE*, що дозволяє заблокованим процесам продовжити виконання.

Зауважимо, що такий підхід також вимагає запису глобально несуперечливого стану, оскільки ніякі вхідні повідомлення не стають частиною стану контрольної точки. Це відбувається тому, що ніякі

повідомлення, наступні за запитом на створення контрольної точки, не вважаються частиною локальної контрольної точки. У той же час, вихідні повідомлення, які надсилаються процесом, для якого створюється контрольна точка, працюючим прикладним програмам, зберігаються в локальній черзі до надходження повідомлення *CHECKPOINT__DONE*.

Удосконаленням цього алгоритму є групова розсилка запиту на створення контрольних точок тільки тим процесам, які залежать від відновлення координатора. Процес вважають залежним від координатора, якщо він одержує повідомлення, яке прямо або опосередковано з певної причини пов'язане з повідомленням, відправленим координатором з моменту створення попередньої контрольної точки, для фіксації станів такого процесу використовують інкрементний знімок стану (*incremental snapshot*).

Для отримання інкрементного знімка стану координатор розсилає запит на створення контрольних точок лише тим процесам, яким він з моменту створення попередньої контрольної точки надсилав повідомлення. Коли процес *P* одержує такий запит, він розсилає його всім іншим процесам, яким з моменту створення попередньої контрольної точки надсилав повідомлення він сам. Процес розсилає запит лише один раз. Коли будуть визначені всі процеси, здійснюється друга групова розсилка, щоб закінчити створення контрольних точок і запустити виконання процесів з того місця, де їх було зупинено.

2.6.3. Протоколювання повідомлень. Характеристичні схеми протоколювання повідомлень

Створення контрольних точок – досить витратна операція, особливо якщо врахувати необхідність запису стану до стійкого сховища, тому слід застосовувати технології зменшення кількості

контрольних точок, не погіршуючі якості виправлення помилок. Одна з найважливіших таких технологій у розподілених системах – це *протоколювання повідомлень* (*message logging*).

Основна ідея, яка лежить в основі протоколювання повідомлень, полягає в тому, що, якщо вдасться відтворити повторну передачу повідомлень, отримують глобальний несуперечливий стан, не відновлюючи його зі стійкого сховища. У цьому разі, починаючи від стану контрольної точки просто відтворюються відправлення, прийом і обробка всіх повідомлень, якими обмінювалися процеси після створення контрольної точки.

Цей спосіб добре працює за умови дотримання вимог кусочно-детермінованої моделі (*piecewise deterministic model*), згідно з якою виконання кожного процесу розбивається на послідовність інтервалів, під час яких відбуваються події, де подією може бути виконання інструкції, відправлення повідомлення тощо. Кожен інтервал у кусочно-детермінованій моделі вважається таким, що починається від недетермінованої події, зокрема від отримання повідомлення. Проте з цього моменту виконання процесу повністю детерміноване. Закінченням інтервалу вважається остання подія з ланцюжка детермінованих перед новою недетермінованою подією.

Інтервал можна повторити детерміновано, починаючи з повторення тієї самої недетермінованої події, тобто, якщо записати в журнал всі недетерміновані події моделі, то дістанемо можливість повністю детерміновано, подія за подією, повторити виконання всього процесу.

Якщо вважати, що журнал повідомлень у разі збою під час виконання процесу необхідно відновити для відновлення глобально несуперечливого стану, то важливо точно знати час запису повідомлень у журнал. Відповідно до використовуваного підходу можна легко

охарактеризувати множину схем протоколювання повідомлень, зосередившись на тому, як вони працюють із процесами-сиротами.

Процес-сирота (*orphan process*) – це процес, стан якого суперечить стану іншого процесу, що спочатку відмовив, а потім відновився.

Характеристичні схеми протоколювання повідомлень. Передбачено, що кожне повідомлення *m* має заголовок, який містить усю інформацію, необхідну для повторної передачі цього повідомлення і правильної його обробки. Так кожен заголовок визначає відправника й одержувача, а також послідовний номер для розпізнавання дублікатів. Окрім того, до заголовка може додаватися вхідний номер, щоб точно встановити момент обробки повідомлення прикладною програмою, яка одержала його.

Повідомлення називають *стійким* (*stable*), якщо воно не може бути втрачено, наприклад, після запису до стійкого сховища. Стійкі повідомлення можуть використовуватися для відновлення за рахунок відтворення їх передачі.

Кожне повідомлення *m* відповідає за набір процесів *DEP(m)*, які залежать від доставки *m*. Зокрема, *DEP(m)* складається з тих процесів, яким має надійти це повідомлення. Якщо інше повідомлення *m′*, яке за певною причиною залежить від доставки повідомлення *m*, надходить до процесу *Q*, то *Q* також входить у набір *DEP(m)*. Відзначимо, що залежність за певною причиною повідомлення *m′* від доставки повідомлення *m* означає, що *m′* надсилається тим самим процесом, який раніше одержав *m* або інше повідомлення, яке за певною причиною залежить від доставки *m*.

Набір *COPY(m)* складається з тих процесів, які містять копію m_i, але не у своїх локальних сховищах. Коли процес *Q* одержує повідомлення *m*, він також стає членом *COPY(m)*. Відзначимо, що *COPY(m)* складається з процесів, які можуть передавати копію *m* для

відтворення передачі. Якщо всі ці процеси відмовили, очевидно, що відтворення передачі *m* стає неможливим.

Визначимо, що ж таке осиротілий процес, для чого припустімо, у розподіленій системі відмовили декілька процесів та нехай *Q* – один зі збережених процесів. Процес *Q* є сиротою, якщо наявне таке повідомлення *m*, яке зумовлює, що *Q* входить до набору *DEP(m)*, тоді як усі процеси з *COPY(m)* відмовили. Інакше кажучи, процес - сирота утворюється в тому разі, якщо він залежить від повідомлення *m*, але немає ніякої можливості повторити передачу цього повідомлення.

Щоб запобігти появі процесів - сиріт, необхідно, щоб у *DEP(m)* не було процесів, які можуть відновитися, якщо відмовлять усі процеси в *COPY(m)*. Система опиняється в такому стані, коли кожен процес, що є членом набору *DEP(m)*, також є членом набору *COPY(m)*, тобто будь-який процес, який залежить від доставки повідомлення *m*, має зберігати копію цього повідомлення.

Розрізняють два основні підходи щодо запобігання появі процесів - сиріт. Перший з них представлений *протоколами песимістичного протоколювання* (*pessimistic logging protocols*), у яких передбачають наявність для кожного нестійкого повідомлення *m* хоча б одного залежного від нього процесу. Інакше кажучи, протоколи песимістичного протоколювання передбачають, що будь-яке нестійке повідомлення *m* доставляється як мінімум в один процес. Відзначимо, що як тільки *m* доставляється, скажімо, у процес *P*, цей процес стає членом набору *COPY(m)*.

Збій може відбутися, коли процес *P* відмовить ще до того, як буде записано повідомлення *m*. У разі песимістичного протоколювання процесу *P* після отримання повідомлення *m* не можна розсилати яких-небудь повідомлень, поки *m* не буде записано у стійке сховище. Таким чином, поки не буде гарантовано можливість повторного розсилання

повідомлення *m*, не зможе з'явитися жоден процес, який залежить від доставки *m* процесу *P*.

Натомість у протоколі *оптимістичного протоколювання* (*optimistic logging protocol*) реальна робота починається після відмови процесу, незважаючи на те, записано повідомлення *m* у стійке сховище чи ні. Зокрема, припустімо, для деякого повідомлення *m* відмовили всі процеси з набору *COPY(m)*. Згідно з оптимістичним підходом будь-який процес-сирота із *DEP(m)* повертається до такого стану, коли він не входить у набір *DEP(m)*.

Песимістичне протоколювання значно простіше за оптимістичне, так що цей спосіб протоколювання повідомлень під час практичної розробки розподілених систем переважає.

2.7. Висновки

1. Відмовостійкість – це важлива складова побудови розподілених систем, яка визначає здатність системи приховувати появу помилок і виправляти їх.
2. Система є відмовостійкою, якщо вона продовжує функціонувати за наявності відмов.
3. Існують різні типи відмов: поломка, яка спричиняє зупинку системи; пропуск даних, який відбувається в тому разі, якщо процес не реагує на вхідні запити; помилка синхронізації, яка спричинена тим, що процес відповідає дуже швидко або дуже повільно; помилки відгуку, такі як некоректний відгук на вхідний запит.
4. Найскладніше обробляти стани, коли у процесі можуть з'явитися помилки будь-якого типу, які називають довільними або візантійськими.

5. Надлишковість – це стандартний спосіб забезпечення відмовостійкості. Якщо надлишковість застосовується до процесів, то процеси поєднують у групи.

6. Процеси у групі тісно взаємодіють між собою для надання деякої послуги. У відмовостійких групах один або більше процесів можуть відмовити, істотно не впливаючи при цьому на доступність послуги, яка реалізується групою, взаємодія всередині групи є високонадійною і у разі забезпечення відмовостійкості підтримує властивості строгої впорядкованості й атомарності.

7. Надійна групова взаємодія, або надійна групова розсилка, може перебувати в різних формах. Поки групи порівняно малі, досягнути надійності можливо. Натомість якщо потрібно підтримувати дуже великі групи, масштабування надійної групової розсилки стає проблематичним. Ключовим моментом у реалізації масштабованості стає зменшення кількості відгуків, що повертають звіт про вдалий (або невдалий) прийом розісланого повідомлення.

8. Більш складно дотримуватись вимоги атомарності. У протоколах атомарної групової розсилки важливо, щоб кожен член групи мав однакове уявлення про те, яким елементам групи доставляється повідомлення. Атомарна групова розсилка має підтримувати модель віртуального синхронного виконання, зокрема дотримуватись меж, усередині яких членство у групах не змінюється, а повідомлення передаються надійно. Повідомлення ніколи не виходять за межі своєї групи.

9. Механізм зміни членства у групі передбачає, що кожен процес має єдиний список членів, що реалізується протоколами підтвердження, найпоширенішим з яких є протокол двофазного підтвердження. У протоколі двофазного підтвердження координатор спочатку перевіряє готовність усіх процесів до виконання однієї операції (тобто до її підтвердження), а на другому етапі розсилає

результат цього голосування. Протокол трифазного підтвердження використовується для того, щоб запобігти відмові координатора без необхідності блокувати всі процеси, які чекатимуть відновлення координатора для досягнення угоди.

10. N-версійне програмування не має альтернативи щодо ефективності у разі виявлення несправностей під час проектування програмного забезпечення гарантоздатних обчислювальних систем, тому воно є потенційно ефективним методом забезпечення відмовостійкості програмного забезпечення. Надзвичайно важливим методом для попередження активізації несправностей у системах довготривалого функціонування є метод прогнозування несправностей на основі ймовірнісно-фізичних моделей відмов (дифузійних розподілів). Перспективність розвитку цих моделей і широке застосування в інформаційних технологіях зумовлено їх ефективністю. Витрати на проектування програмного забезпечення, прогнозування несправностей та попередження відмов мають високу окупність порівняно з втратами часу на виявлення помилки та продуктивності у разі обробки несправності й відновлення системи після усунення вже наявної помилки.

11. Відновлення у відмовостійких системах незмінно організовують на основі регулярного збереження інформації про стан системи. Робота з контрольними точками повністю розподілена. Створення контрольної точки є досить витратною операцією. Для підвищення продуктивності багатьох розподілених систем поєднують контрольні точки з протоколюванням повідомлень. Завдяки протоколюванню взаємодії між процесами після відмови системи можна відтворити перебіг її функціонування.

2.8. Запитання і завдання для самоконтролю

1. Розкрийте поняття «відмовостійкість» розподілених систем.
2. Назвіть вимоги до розподілених систем.
3. Які бувають види та моделі відмов?
4. Що таке маскування помилок?
5. Поясніть поняття «відмовостікість» процесів.
6. Які нявні способи реалізації відмовостійкості програмного забезпечення?
7. У чому полягає суть N–версійного програмування?
8. Назвіть групи взаємодіїї процесів та поясніть принцип їх формування.
9. Як процес може вступати до групи?
10. Поясніть, що таке реплікація?
11. Який принцип закладено у крізне передавання?
12. Що таке семантика RPC?
13. Назвіть і поясніть помилки, які можуть виникати в системах RPC.
14. Що спільного в помилках, які можуть виникати в системах RPC?
15. Поясніть поняття «масштабованість» надійної групової розсилки.
16. Що таке ієрархічне та неієрархічне керування зворотним зв’язком?
17. У чому суть атомарної групової розсилки?
18. Які бувають види атомарної групової розсилки?
19. Поясніть поняття «розподілене підтвердження».
20. Які відмінності між двофазним та трьохфазним підтвердженням?
21. Розкрийте поняття «відновлення».
22. Що таке стійкі сховища?
23. Поясніть процес створення контрольних точок.
24. Охарактеризуйте протоколювання повідомлень.

3. ЗАХИСТ ІНФОРМАЦІЇ

Системи захисту в розподілених системах можна поділити на дві незалежні частини. Одна з них – це зв'язок між користувачами або процесами, можливо, розташованими на різних машинах.

Друга частина систем захисту – це авторизація, яка гарантує, що процеси одержать лише ті можливості доступу до ресурсів розподіленої системи, на які мають право. Авторизацію і контроль доступу можна розглядати спільно. Захищені канали і засоби контролю доступу потребують механізмів для роботи з криптографічними ключами, а також механізмів додавання користувачів у систему і видалення їх із неї.

Захист у розподілених системах пов'язаний з поняттям «надійність» (*dependability*), тобто надійною розподіленою системою вважають таку, службам якої виправдано довіряють.

Цілісність (*integrity*) – це характеристика, яка визначає, що зміни можуть бути внесені в систему тільки авторизованими особами або процесами. Несанкціоновані зміни в захищеній розподіленій системі мають виявлятися і усуватися. Основні частини розподіленої системи – це апаратура, програми і дані.

Розглянемо такі можливі ситуації, які виникають в розподілених системах та вимагають захисту, як *перехоплення* (*interception*), *переривання* (*interruption*), *модифікація* (*modification*), *підроблення* (*fabrication*).

Перехоплення – це ситуація, коли неавторизований агент одержує доступ до служб або даних. Типовий приклад перехоплення: коли зв'язок між двома агентами підслуховує хтось третій.

Прикладом **переривання** може бути ушкодження або втрачання файлу. Звичайне переривання пов'язують з такою ситуацією, коли служби або дані стають недоступними, знищуються, їх неможливо

використати тощо. У цьому сенсі атаки «відмова в обслуговуванні» (*denial of service*), за яких хтось навмисно намагається зробити визначену службу недоступною для інших, – це загроза захисту, яку класифікують як переривання.

Модифікації містять неавторизовані зміни даних або фальсифікацію служб для того, щоб вони не відповідали своєму оригінальному призначенню. Прикладом модифікації є перехоплення повідомлень з подальшим змінюванням переданих даних, фальсифікація входів у бази даних і змінювання програм для того, щоб потай відслідковувати дії користувачів.

Підробленню відповідає ситуація, коли створюються додаткові дані або виконується діяльність, неможлива за нормальних умов. Так, зловмисник може спробувати додати записи у файл паролів або базу даних. Окрім того, іноді вдається ввійти в систему, відтворивши відправлення раніше надісланого повідомлення.

Переривання, модифікація і підроблення можна розглядати як форми фальсифікації даних.

Просто констатувати, що система має бути здатною протистояти всіляким загрозам захисту – це недостатньо, щоб побудувати захищену систему. Необхідно описати вимоги до захисту, які називають *правилами захисту* (*security policy*) та які точно описують дозволені й заборонені дії для системних сутностей, до яких відносять користувачів, служби, дані, машини тощо. Після складання правил захисту потрібно обґрунтувати вибір *механізмів захисту* (*security mechanisms*), за допомогою яких реалізуються ці правила. Найбільш важливі з механізмів такі: шифрування (*encryption*), аутентифікація (*authentication*), авторизація (*authorization*), аудит (*auditing*).

Шифрування – фундамент захисту в розподілених системах, який трансформує дані в дещо, чого зловмисник не взмозі зрозуміти; це засіб реалізації конфіденційності. Крім того, шифрування дозволяє

перевірити, чи не змінювалися дані, даючи можливість контролювати цілісність даних.

Аутентифікація використовується для перевірки заявленого імені користувача, клієнта, сервера та ін. До початку роботи з клієнтом служба має визначити справжність клієнта, для чого потрібно його аутентифікувати. Зазвичай користувачі аутентифікують себе за допомогою пароля, однак існують й інші способи аутентифікації клієнта.

Після того як клієнт аутентифікований, необхідно перевірити, чи має він право на виконання запитуваних дій, наприклад отримати доступ до бази даних з медичною інформацією. Залежно від того, хто працює з базою, йому може бути дозволено читати записи, модифікувати визначені поля записів або додавати і видаляти записи.

Засоби аудиту використовують для контролю за тим, що робить клієнт і як він це робить. Хоча засоби аудиту не захищають від загроз захисту, журнали аудиту постійно застосовують для аналізу «дір» у системах захисту з подальшим уживанням заходів проти порушників, тому порушники намагаються не залишати яких-небудь слідів, що зрештою дозволять їх розкрити. Саме завдяки протоколюванню доступу вдається протистояти «хакерству».

3.1. Розробка механізмів захисту

У розподілену систему мають бути вбудовані механізми захисту, за допомогою яких можна реалізувати різні правила захисту, при цьому варто враховувати три важливих ознаки: фокус керування, багаторівневу організацію механізмів захисту і простоту.

3.1.1. Фокус керування

Організувати захист прикладних програм (зокрема розподілених) можна, використовуючи три основних підходи. Перший підхід – це захист, безпосередньо асоційований з прикладною програмою, пов’язаною з доступом та обробкою даних, тобто незалежно від операцій, які можуть здійснюватися нею з елементами даних, має бути збережена цілісність даних. Зазвичай такий захист використовують в системах баз даних, тоді заздалегідь визначають різні обмеження цілісності, які потім автоматично перевіряють у процесі кожної модифікації елемента даних.

Другий підхід – це захист, який застосовує точну вказівку щодо того, хто і як може використати операції доступу до даних або ресурсів. У цьому разі фокус керування тісно пов'язаний з механізмами контролю доступу. Цей підхід також використовується з метою деталізації керування доступом.

Приклад. У системі, побудованій на основі об'єктів, для кожного з методів, доступ до яких відкривається клієнтам, потрібно зазначити, який саме клієнт має право до яких методів та до яких об’єктів звернутися. Методи контролю доступу можуть застосовувати до інтерфейсу, який надає об'єкт або властивий об'єкту.

Третій підхід – зосередити увагу безпосередньо на користувачеві, вживаючи заходів, щоб доступ до прикладного програмного забезпечення або його ресурсів мали тільки визначені люди, незалежно від операцій, які вони збираються виконати.

Приклади. Розглянемо перший випадок - банківська база даних може бути захищена за рахунок закриття доступу до неї всім, крім вищого управлінського персоналу. Другий випадок, коли у багатьох університетах доступ до визначених даних і прикладних програм дозволено лише викладачам і персоналу, натомість студентам доступ до них закритий.

У цих двох випадках керування насправді зводять до визначення *ролей* (*roles*) користувачів. Після підтвердження відповідної ролі їй

надається або забороняється доступ до відповідних ресурсів. Таким чином, процес розробки системи захисту полягає, зокрема, у тому, щоб визначити ролі, які можуть знадобитися користувачам, і надати механізми керування доступом на основі списків ролей.

3.1.2. Багаторівнева організація механізмів захисту

Під час розробки систем захисту важливо визначитись, скільки рівнів повинні мати механізми захисту. Рівень у цьому контексті відповідає логічній організації системи.

Комбінуючи багаторівневі структури інформаційно-телекомунікаційних мереж і розподілених систем, отримуємо схему, подану на рис. 3.1.

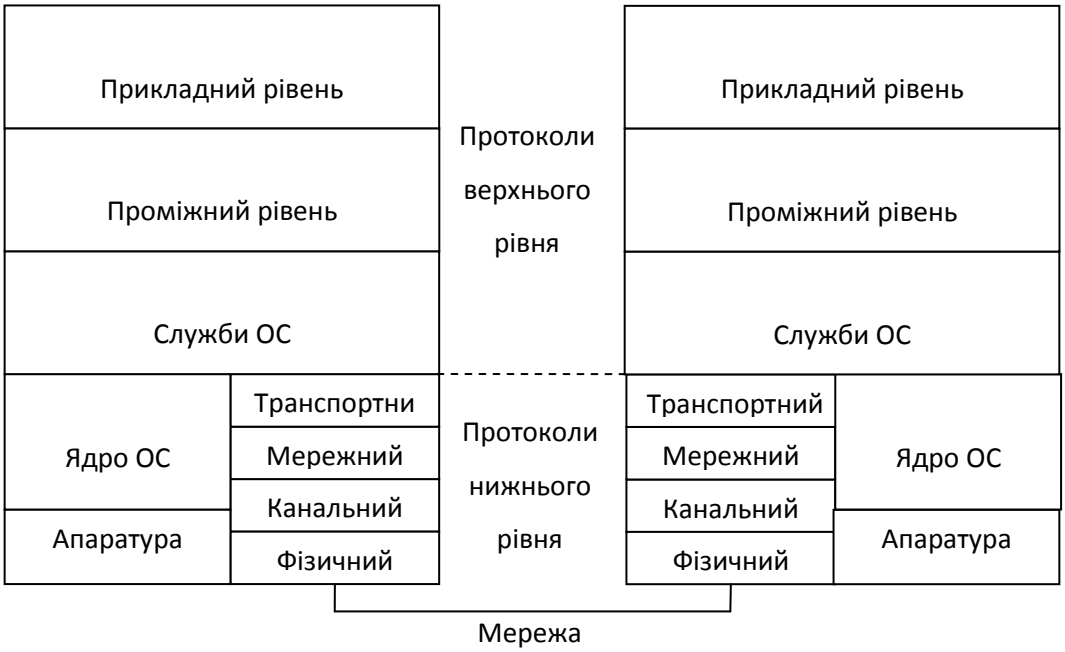


Рис. 3.1. Логічна багаторівнева організація розподілених систем

На рис. 3.1 служби загального призначення відділено від комунікаційних служб. Цей поділ дуже важливий для розуміння розподілу за рівнями механізмів захисту розподілених систем і, зокрема, для уявлення про довіру, причому різниця між поняттями «довіра» і «захист» є суттєвою. Система може бути або не бути

захищеною, але думка клієнта про те, що система захищена – це питання довіри. Рівень, на якому розміщують механізм захисту, залежить від довіри клієнта до захисту його служб.

3.1.3. Розподіл механізмів захисту

Залежності між службами, що вимагають довіри, призводять до виникнення поняття «довірена обчислювальна база» (*Trusted Computing Base,* TSB), яка є набором усіх механізмів захисту розподіленої системи, необхідних для реалізації правил захисту, при цьому чим менший TSB, тим краще. TSB у розподіленій системі може містити локальні операційні системи різних хостів. Якщо розподілена система побудована на основі проміжного рівня (надбудови над наявною мережною операційною системою), її захист залежатиме від захисту базових локальних операційних систем.

Розподілені системи, побудовані з використанням технологій проміжного рівня, вимагають довіри до локальної операційної системи, під керуванням якої вони працюють. Якщо такої довіри немає, то частина функціональності локальної операційної системи має бути вбудована в саму розподілену систему. Враховуючи, що в операційній системі є мікроядро, більшість служб виконані у вигляді звичайних процесів користувача, то, наприклад, стандартна файлова система може бути повністю замінена іншою, особливо пристосованою до специфічних потреб розподіленої системи, зокрема до різних механізмів захисту. У такому разі доцільно відділити служби захисту від інших видів служб за рахунок розподілу цих служб по різних машинах відповідно до того ступеня захисту, який їм необхідний.

Приклад. Для захищеної розподіленої файлової системи можна відокремити файловий сервер від клієнтів, установивши сервер на машину з довіреною операційною системою, на якій наявна також окрема захищена файлова система.

Натомість, клієнти і їх прикладні програми можуть бути розміщені на машинах, що не викликають довіри.

3.2. Криптографія

У захисті розподілених систем дуже важливу роль відіграє криптографія. Головну ідею, яку покладено в основу засобів криптографії, ілюструє наступне: відправник *S*, який хоче переслати повідомлення *T* одержувачеві *R*, щоб убезпечити повідомлення від загроз захисту, спочатку шифрує його, перетворюючи в незрозуміле повідомлення *T'*, після чого посилає його одержувачеві *R*, котрий, у свою чергу, розшифровує отримане повідомлення й одержує оригінал, тобто повідомлення *T*.

Шифрування і розшифрування здійснюють криптографічними методами з використанням ключів. Вихідна форма повідомлення, яка надсилається, або *простий текст* (*plaintext*), позначимо як *P*, його зашифрований варіант, відомий як *шифрований текст* (*ciphertext*), позначимо як *C*.

Щоб описати різні протоколи, використовувані для побудови служб захисту розподілених систем, бажано ввести позначення для звичайного тексту, шифрованого тексту і ключів. Відповідно до загальних положень використовуватимемо вираз *C = EK(P)* для опису того, що шифрований текст *C* було отримано шифруванням простого тексту *P* з використанням ключа *K*. Так само *P = DK(C)* позначає операцію розшифрування шифрованого тексту *C* із використанням ключа *K*, за допомогою якого можна отримати простий текст *P*.

Під час передачі повідомлення у вигляді шифрованого тексту C, це повідомлення має бути захищеним від таких трьох різних типів атак:

1. Зловмисник може перехопити повідомлення так, що про це не довідаються ні відправник, ні одержувач. Зрозуміло, якщо овідомлення,

яке передається, зашифровано таким чином, що його неможливо розшифрувати, не маючи відповідного ключа, то перехоплення марне: зловмисник зможе побачити тільки незрозумілі дані. (До речі, самого факту передачі повідомлень може іноді бути достатньо для зловмисника, щоб зробити певні висновки).

Приклад. Якщо під час світової кризи обсяг вхідного трафіку центрального офісу глобальної торгової компанії раптово знижується майже до нуля, а обсяг трафіку підпорядкованої компанії, яка працює з клієнтами, відповідно підвищується, то це може виявитися дуже корисною інформацією.

2. Зловмисник може змінити повідомлення. Змінити грамотно зашифрований текст набагато складніше, ніж простий, оскільки для внесення в нього осмислених змін зловмисник повинен спочатку розшифрувати повідомлення, а також коректно зашифрувати його, інакше одержувач може помітити, що повідомлення підроблено.

3. Зловмисник може додати шифроване повідомлення в систему комунікації зі спробою змусити R повірити, що це повідомлення надійшло від S. Криптографічні системи розділяють на системи з однаковими і різними ключами шифрування і дешифрування.

У разі *симетричної криптосистеми* (*symmetric cryptosystem*) для шифрування і розшифрування повідомлення використовується один ключ: $P = DK(EK(P))$. Симетричні криптосистеми також називають системами із секретним, або загальним, ключем, оскільки відправник і одержувач повинні спільно використати той самий ключ. Для гарантування захисту загальний ключ має бути секретним, тобто бути відомим тільки відправнику та одержувачу. Для такого ключа, поділюваного між *A* і *B*, використовують позначення $K_{a,в}^{+}$.

В *асиметричній криптосистемі* (*asymmetric cryptosystem*) ключі шифрування і розшифрування різні, але разом утворюють унікальну пару. Існує окремий ключ шифрування K_e й окремий ключ розшифрування, K_d, так що: $P = DK_d(EK_e(P))$.

Один ключ асиметричної криптосистеми є закритим, другий – відкритим. З цієї причини асиметричні криптосистеми також називають системами з *відкритим ключем* (*public-key systems*), тому використовуватимемо позначення K_a^+ для позначення відкритого ключа, що належить *A*, а Ka^- – для відповідного закритого ключа.

Який з ключів (шифрувальний або дешифрувальний) буде відкритим, залежить від того, як їх використовують.

Приклад. Якщо *Користувач 1* хоче надіслати *Користувачу 2* конфіденційне повідомлення, то він використовуватиме для шифрування повідомлення відкритий ключ *Користувача 2*. Оскільки *Користувач 2* – єдиний, хто має закритий ключ розшифрування, він буде також єдиним, хто зможе розшифрувати повідомлення.

Припустімо, *Користувач 2* хоче впевнитися, що повідомлення, яке він щойно одержав, дійсно відправлене *Користувачем 1*. У цьому разі *Користувач 1* може застосувати для шифрування повідомлення свій закритий ключ. Якщо *Користувач 2* у змозі успішно розшифрувати повідомлення, використовуючи відкритий ключ *Користувача 1* (і розшифрований текст повідомлення містить достатньо інформації, щоб переконати *Користувача 2*), він знає, що повідомлення було надіслано *Користувачем 1*, оскільки ключ розшифрування однозначно пов'язаний з ключем шифрування.

Ще один варіант застосування криптографії в розподілених системах – це використання *хеш-функції H* (*hash functions*), яка приймає на вході повідомлення *m* довільної довжини і створює рядок бітів *h* фіксованої довжини: $h = H(m)$.

Хеш *h* можна порівняти з додатковими бітами, які вводять до повідомлення в комунікаційній системі для виявлення помилок, наприклад, у разі *циклічного надлишкового кодування* (*cyclic redundancy check, CRC*). Хеш-функції, використовувані у криптографічних системах, мають такі властивості:

1. Це однобічні функції (*one-way functions*), у разі застосування яких неможливо обчислити вхідне повідомлення *m*, якщо відомий результат роботи однобічної функції *h*, на відміну від того, що обчислення *h* за *m* є досить нескладним.

2. Однобічні функції мають низьку стійкість до колізій (*weak collision resistance*), тобто якщо задано вихідне повідомлення *m*, якому відповідає результат *h = H(m)*, то неможливо обчислити інше повідомлення *m'*, зокрема *H(m) = H(m')*. Криптографічна хеш-функція також має високу стійкість до колізій (*strong collision resistance*). Це означає, що якщо задано тільки хеш-функцію *H*, то неможливо обчислити будь-які два різних вхідних значення *m* та *m'*, так, щоб *H(m) = H(m')*.

Подібні властивості застосовуються до будь-якої функції шифрування *E* і використовуваних ключів. Для будь-якої функції шифрування *E* не має бути можливості обчислити використовуваний ключ *K*, маючи простий текст *P* і відповідний йому зашифрований текст *C = EK(P)*. Також аналогічно стійкості до колізій, за наявності простого тексту *P* і ключа *K* неможливо обчислити інший підходящий ключ *K'*, такий, щоб *EK(P) = EK' (P)*.

Коротко розглянемо три показові криптографічні алгоритми, передусім узагальнивши використовувану нотацію й абревіатури (табл. 3.1).

Таблиця 3.1. Нотації та їх інтерпретація

Нотація	**Опис**
$K_{а,в}$	Секретний ключ, спільно використовуваний *A* і *B*
Ka^{+}	Відкритий ключ *A*
Ka^{-}	Закритий ключ *A*

Як приклад криптографічного алгоритму розглянемо стандарт шифрування даних *(Data Encryption Standard, DES*), що використовується в симетричних криптосистемах. Алгоритм *DES* створювали для роботи із 64-бітовими блоками даних. Згідно з алгоритмом блок перетворюють в зашифрований (64-бітний) блок за 16 кроків, на кожному з яких застосовується власний 48-бітний ключ шифрування. Кожний із цих 16 ключів утворюється з 56-бітного головного ключа, як показано на рис. 3.2, *а*. Перед тим як над вихідним

блоком почнуть виконувати його 16 циклів шифрування, здійснюється його первинне перетворення, а після шифрування - перетворення, обернене первинному, результатом чого є зашифрований блок.

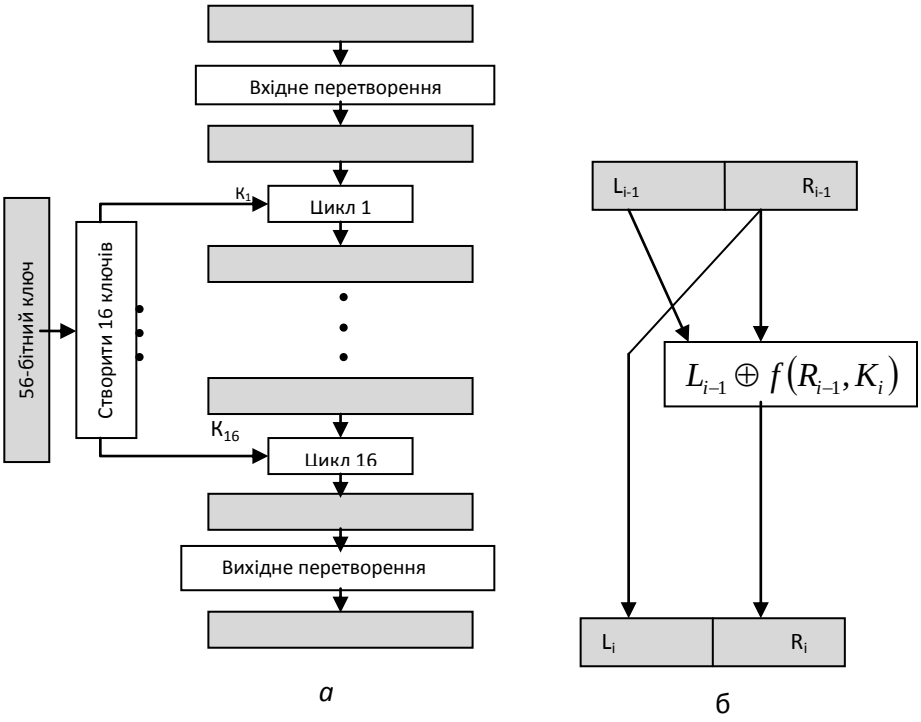


Рис. 3.2. Алгоритм DES (*а*); схема одного циклу шифрування (*б*)

Кожен цикл шифрування бере як вихідні дані 64-бітний блок з попереднього циклу шифрування *i - 1*, рис. 3.2, *б*. 64 біти розбиваються на ліву частину L_{i-1} і праву частину R_{i-1}, по 32 біта кожна. Права частина в наступному циклі використовується як ліва, і навпаки.

Основна робота виконується у функції *f*. Ця функція приймає як вихідні дані 32-бітний блок R_{i-1} і 48-бітний ключ K_i, а потім генерує 32-бітний блок, що піддається операції «*що виключає АБО*» *(XOR)* із блоком L_{i-1}, породжуючи R_i. Функція, що перетворює, спочатку розширює R_{i-1} до 48 біт, а потім проводить над ним операцію «що виключає *АБО*» із ключем K_i. Результат ділиться на вісім частин по шість біт. Кожна частина проходить потім через різні *S*-бокси (*S-boxes*), які становлять операції заміни кожної з 64 можливих 6-бітних комбінацій на одну з 16-ти можливих 4-х бітних комбінацій. Вісім

складових, що вийшли до коду, по 4 біти поєднуються в одне 32-бітове значення і перетворюються далі.

48-бітний ключ K_i породжується з 56-бітного головного ключа в такий спосіб. Спочатку головний ключ перетворюється і ділиться на дві 28-бітних половини. У кожному циклі перша половина зміщується на один або два біти вліво, після чого з неї виділяються 24 біти. Разом з 24-ма бітами з другої зміщеної половини вони утворюють 48-бітний ключ. Детальний опис одного циклу шифрування подано на рис. 3.3.

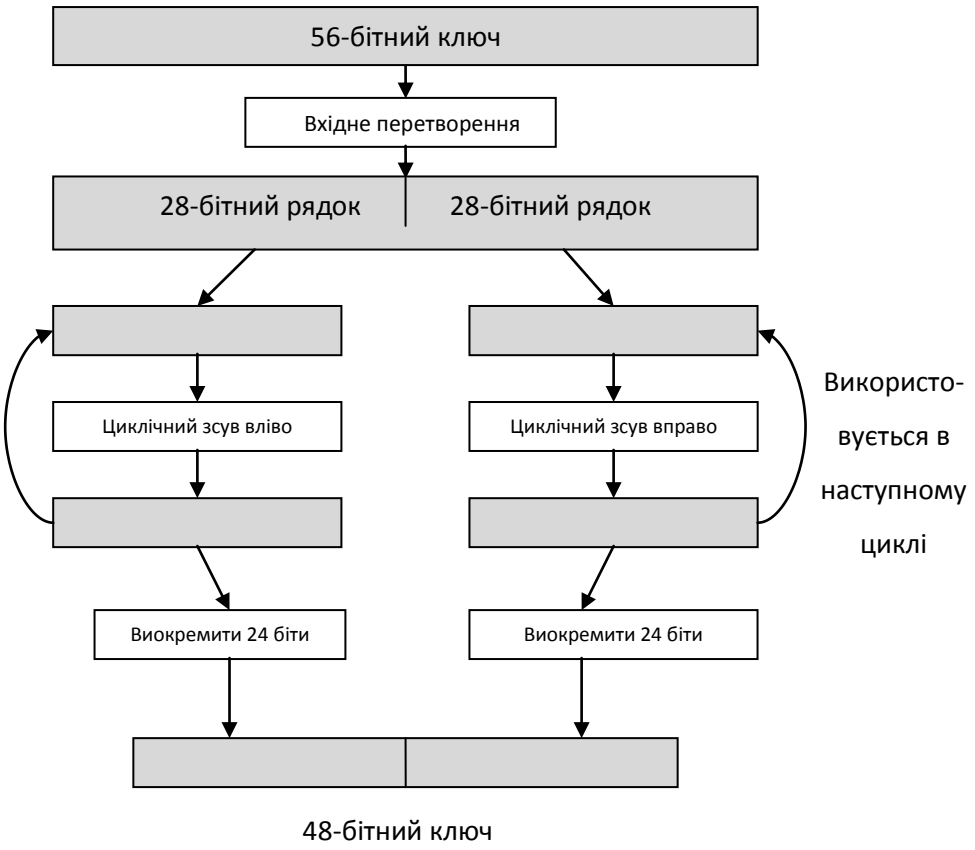


Рис. 3.3. Техніка циклічної генерації ключа в алгоритмі *DES*

Алгоритм DES досить простий, але його нелегко зламати, використовуючи аналітичні методи, для його злому простий підбір ключа є значно ефективнішим. Нині надійно захищає від злому «потрійне» використання алгоритму *DES* у спеціальному режимі шифрування – розшифрування – шифрування з різними ключами.

Аналітичні атаки з метою отримання ключів, зашифрованих за допомогою алгоритму *DES*, сильно ускладнює те, що структура цього

алгоритму ніколи не була повністю описана у відкритій документації. Однак, застосування нестандартних *S*-боксів значно спрощує злом алгоритму.

Алгоритм *DES* уже багато років використовується як стандартна технологія шифрування, але нині відбувається процес його заміни алгоритмом *Риндаля* (*Rijndael*) із блоками довжиною 128 біт, який є досить швидким, тому його можна реалізувати навіть у смарт-картах.

Як другий приклад криптографічного алгоритму розглянемо широко використовувану систему з відкритим ключем – *RSA*, названу на честь її винахідників – Рівеста (*Rivest*), Шаміра (*Shamir)* і Альдемана (*Aldeman*). Головною ідеєю, покладеною в основу вирішення проблеми захисту в алгоритмі RSA, є те, що на сьогодні немає ефективного методу визначення простих множників великих чисел. Будь-яке ціле число можна записати у вигляді добутку простих чисел.

Приклад. Число 2100 може бути записане як 2100 = 2*2*3*5*5*7.

Таким чином, 2, 3, 5 і 7 є простими множниками числа 2100.

У RSA відкритий і закритий ключі створюються з дуже великих простих чисел. Злом RSA еквівалентний виявленню цих чисел. Нині таку задачу розв'язати неможливо, незважаючи на те, що математики працюють над нею вже кілька сторіч.

Створення відкритих і закритих ключів відбувається в чотири етапи:

1) обираються два більших простих числа, p і q;
2) обчислюється їх добуток $n = p \times q$; $z = (p - 1) \times (q – 1)$;
3) обирається число d, взаємно просте з z;
4) обчислюється число e, таке, що $e \times d = 1 \bmod z$.

Одне із чисел, наприклад d, можна згодом використати для розшифрування, а e – для шифрування. Тільки одне із цих двох чисел буде відкритим, яке саме – залежить від використовуваного алгоритму.

Користувач 1 хоче, щоб повідомлення, яке він надсилає *Користувачу 2*, було конфіденційним, тобто хоче упевнитися в тому, що ніхто, крім *Користувача 2*, не зможе перехопити і прочитати його повідомлення. Алгоритм RSA розглядає будь-яке повідомлення m як рядок бітів. Кожне повідомлення спочатку розбивається на блоки фіксованої довжини, і кожен черговий блок m_i подається у вигляді двійкового числа, яке перебуває в інтервалі $0 \le m_i < n$.

Для шифрування повідомлення m відправник обчислює для кожного блока m_i значення $c_i = me_i \ (mod \ n)$, що і відправляється одержувачеві. Розшифрування одержувачем відбувається за допомогою обчислення $m_i = cd_i (mod \ n)$. Відзначимо, що для шифрування необхідні e і n, у той час як для розшифрування – d і n.

Порівняємо RSA із симетричними криптосистемами, зокрема DES. Для алгоритму RSA характерний недолік – складність обчислень, через що розшифрування повідомлень, зашифрованих за алгоритмом RSA, займає в 100 ÷ 1000 разів більше часу, ніж розшифрування повідомлень, зашифрованих за алгоритмом DES. Точний показник складності обчислень залежить від реалізації алгоритмів, тому багато криптографічних систем використовують алгоритм RSA тільки для безпечного обміну загальними секретними ключами, не застосовуючи цього способу для шифрування «звичайних» даних.

Криптографічний алгоритм MD5, який широко використовується, – це хеш-функція для обчислення 128-бітних дайджестів повідомлень (*message digests*) фіксованої довжини з двійкових вихідних рядків довільної довжини. Спочатку вихідний рядок доповнюється до загальної довжини 448 біт (за модулем 512), після чого до нього додається довжина вихідного рядка у вигляді 64-бітового цілого числа, у результаті чого вихідні дані перетворяться в набір 512-бітних блоків.

Структуру алгоритму наведено на рис. 3.4. Починаючи з визначеного постійного 128-бітового значення, алгоритм містить k фаз,

де k – кількість 512-бітних блоків, які отримано із доповненого відповідно до алгоритму повідомлення. У ході кожної фази з 512-бітного блока даних і 128-бітного дайджесту (*message digests*), обчисленого на попередній фазі, розраховується новий 128-бітний дайджест. Фаза алгоритму MD5 має чотири цикли обчислень, у кожному з яких використовується одна з таких чотирьох функцій:

1) $F(x, y, 2) = (x \text{ AND } y) \text{ OR } ((\text{NOT } x) \text{ AND } z)$;
2) $G(x, y, z) = (x \text{ AND } z) \text{ OR } (y \text{ AND } (\text{NOT } z))$;
3) $H(x,y,z) = x \text{ XOR } y \text{ XOR } z$;
4) $I(x,y,z) = y \text{ XOR } (x \text{ OR } (\text{NOT } z))$.

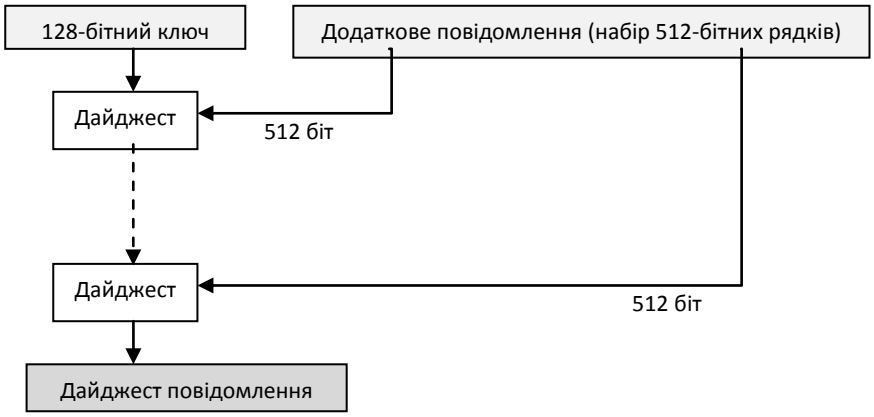


Рис. 3.4. Структура MD5

Кожна із цих функцій працює з 32-бітними змінними x, y і z. Щоб проілюструвати, як використовуються ці функції, розглянемо 512-бітний блок b доповненого повідомлення на фазі k. Блок b розділяється на шістнадцять 32-бітних підблоків b_o, b_1, ... , b_{15}. У першому циклі для зміни чотирьох змінних (назвемо їх p, q, r і s відповідно) у ході 16 –ти ітерацій використовується функція F. Ці змінні переносяться на черговий цикл, а після закінчення цієї фази передаються на наступну. Всього існує 64 заздалегідь визначених констант C_i. Для зазначення циклічного зсуву вліво використовується запис $x << n$: біти в x зміщуються на n позицій, причому біти, які опинилися зліва від межі числа, додаються праворуч.

У другому циклі аналогічно використовується функція G, а H і I – відповідно у третьому і четвертому циклах. Кожен крок містить у собі 64 ітерації, і в черговій фазі використовуються обчислені на попередній фазі значення p, q, r і s.

3.3. Захищені канали

Побудова захищеної розподіленої системи передбачає дві основні складові: побудову захищеного зв'язку між клієнтом і сервером та спосіб авторизації.

Захищений зв'язок вимагає аутентифікації взаємодіючих сторін, що гарантує цілісність повідомлень, а можливо, й їх конфіденційність.

Спосіб авторизації визначає яким чином серверу, який одержав запит від клієнта, розпізнати факт авторизації клієнта для передачі цього запиту на обробку. Авторизація потрібна, щоб здійснювати керований доступ до ресурсів.

Захист зв'язку між клієнтами і серверами є організацією між з’єднаними сторонами захищеного каналу (*secure channel*). Захищені канали оберігають відправника й одержувача від перехоплення, модифікації та підроблення повідомлення. Зазвичай немає потреби вводити захист від переривання зв'язку, бо це не призведе до отримання повідомлення зловмисником. Захист повідомлень від перехоплення здійснюється за рахунок гарантованої конфіденційності: захищені канали гарантують, що повідомлення в них не можуть перехопити зловмисники. Захист повідомлень від модифікації або підроблення забезпечується за допомогою протоколів взаємної аутентифікації та цілісності повідомлень.

3.3.1. Аутентифікація повідомлень

Для захисту повідомлень необхідно використовувати одночасно як їх аутентифікацію, так і цілісність. У багатьох протоколах їх комбінація працює приблизно в такий спосіб. Припустімо, *Користувачу 1* і *Користувачу 2* потрібно поспілкуватися, і *Користувач 1* проявляє ініціативу з організації каналу. Він починає цей процес із надсилання повідомлення *Користувачу 2* або довіреній третій особі, яка допоможе встановлювати канал. Коли канал встановлено, *Користувач 1* упевнений, що він спілкується з *Користувачем 2*, а *Користувач 2* упевнений, що спілкується з *Користувачем 1*. Тепер може початися обмін повідомленнями.

Щоб надалі забезпечити цілісність повідомлень, якими вони обмінюватимуться після аутентифікації, застосовується криптографія із секретним ключем з використанням сеансових ключів. *Сеансовим* (*session key*) називають загальний ключ, який застосовують для шифрування повідомлень для дотримання їх цілісності, а також, можливо, конфіденційності. Цей ключ потрібен тільки протягом часу існування каналу і скасовується (або знищується з дотриманням заходів безпеки) після закриття каналу.

3.3.2. Цілісність і конфіденційність повідомлень

Крім аутентифікації, захищений канал має також гарантувати цілісність і конфіденційність повідомлень. Під цілісністю повідомлень розуміють захищеність повідомлень від зміни, а конфіденційність означає, що повідомлення не можуть бути перехоплені та прочитані сторонніми особами. Конфіденційність легко реалізується за допомогою звичайного шифрування повідомлень перед їх відправленням. Шифрування може здійснюватися як секретним

ключем, спільно використовуваним відправником і одержувачем, так і відкритим ключем одержувача.

3.3.3. Цифрові підписи

Цілісність повідомлень часто не обмежується лише передачею повідомлень безпечним каналом. Розглянемо ситуацію, коли *Користувач 2* продає *Користувачу 1* якісь цифрові фотографії до колекції за певну суму. Укладаючи угоду електронною поштою *Користувач 1* надсилає *Користувачу 2* повідомлення, яке підтверджує, що він купує фотографії за цю суму. Окрім аутентифікації для підтримання цілісності повідомлення важливо дотримуватися ще як мінімум двох умов.

Користувач 1 повинен впевненитися, що *Користувач 2* не зможе навмисно змінити обумовлену суму, зазначену в його повідомленні, на більшу, і ствердити, що йому обіцяно більшу суму.

Користувач 2 повинен впевненитися, що *Користувач 1* не зможе відмовитися від своєї пропозиції після відправлення повідомлення, наприклад, якщо передумає.

Ці дві умови можна забезпечити, якщо *Користувач 1* поставить свій *цифровий підпис* (*digital signature*) під цим документом, однозначно пов'язавши його зі змістом листа. Завдяки унікальному зв'язку між повідомленням і підписом під ним внесення змін у повідомлення не залишиться непоміченим. Крім того, якщо справжність підпису *Користувача 1* може бути підтверджено, то йому не вдасться надалі заперечувати того факту, що повідомлення підписав саме він.

Існує кілька способів поставити під документом цифровий підпис. Один з найбільш популярних варіантів – використання криптосистем з відкритим ключем, зокрема RSA (рис. 3.5). Коли *Користувач 1*

надсилає повідомлення *Користувачу 2*, він шифрує його своїм закритим ключем $КА^-$ і пересилає. Якщо він хоче одночасно зберегти зміст листа в таємниці, він може скористатися також і відкритим ключем *Користувача 2* і надіслати $КВ^+(m, КА^-(m))$, поєднавши в листі повідомлення *m* і його екземпляр, зашифрований *Користувачем 1*.

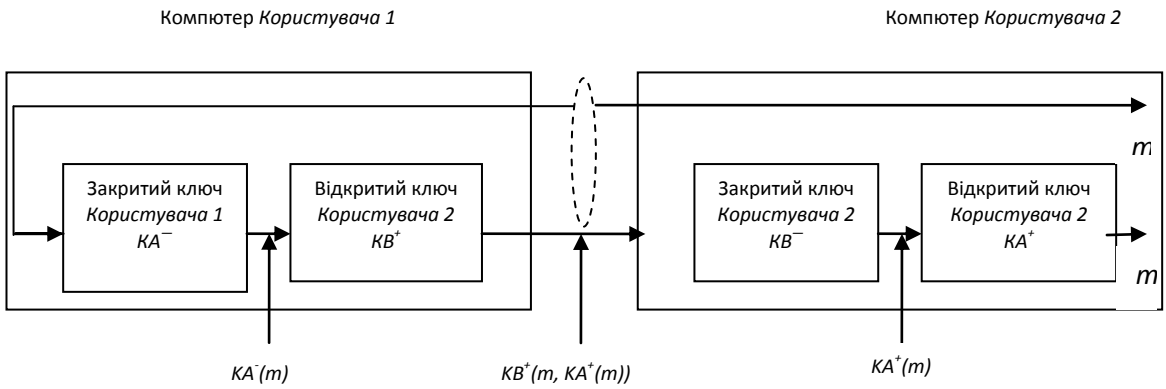


Рис. 3.5. Цифровий підпис повідомлення і шифрування з відкритим ключем

Коли повідомлення дійде до *Користувача 2*, він зможе розшифрувати його, використовуючи відкритий ключ *Користувача 1*. Якщо він буде впевнений, що відкритий ключ дійсно належить *Користувачеві 1*, то розшифрування підписаної версії повідомлення *m* і успішне порівняння його з вихідною версією повідомлення *m* означатиме, що воно дійсно надійшло від *Користувача 1*. *Користувач 1* захищений від будь-яких навмисних змін *m*, внесених *Користувачем 2*, оскільки *Користувач 2* завжди може підтвердити, що модифіковану версію *m* також підписав *Користувач 1*. *Користувач 2* зберігатиме підписану версію *m*, щоб захистити себе від спроб *Користувача 1* відмовитися від своєї пропозиції.

Цій схемі взаємодії властиві певні проблеми, хоча сам протокол абсолютно коректний. По-перше, підпис *Користувача 1* дійсний тільки доти, поки його закритий ключ утримується в секреті. Якщо *Користувач 1* захоче переглянути умови угоди вже після відправлення

Користувачеві 2 підтвердження, то йому достатньо буде оголосити, що до надсилання підтвердження в нього було вкрадено закритий ключ.

По-друге, проблема виникає в тому разі, якщо *Користувач 1* вирішить змінити свій закритий ключ. Зміна ключа, виконувана час від часу, зазвичай сприяє підвищенню захисту, однак якщо *Користувач 1* змінив свій ключ, то договір, відправлений *Користувачу 2*, втрачає чинність. У подібних випадках використовують централізовану службу авторизації, яка відслідковуватиме зміни ключів і відмітки часу підписання повідомлень.

Недоліком такої схеми є те, що *Користувач 1* шифрує все повідомлення своїм закритим ключем, що для розшифрування потребує великої кількості обчислювальних ресурсів (або навіть виявитися нездійсненним, якщо припустити, що повідомлення інтерпретується як двійкове число з обмеженою максимальною довжиною). Для запобігання цьому однозначно зв'язують підпис із конкретним повідомленням, а також використовують схему, яка застосовує дайджести повідомлень.

Дайджест повідомлення – це рядок бітів фіксованої довжини h, який обчислюється з повідомлення довільної довжини m за допомогою криптографічної хеш-функції H. Якщо m змінюється, то утворюється інше повідомлення m'. Його хеш $H(m')$ відрізнятиметься від первинного рядка $h = H(m)$, так що внесені зміни буде легко виявити.

Щоб підписати повідомлення, *Користувач 1* повинен спочатку обчислити дайджест повідомлення, а потім зашифрувати його своїм закритим ключем (рис. 3.6). Зашифрований дайджест пересилається *Користувачу 2* разом з повідомленням. Відзначимо, що саме повідомлення пересилається відкритим текстом і його може прочитати хто завгодно, тому для конфіденційності повідомлення можна також зашифрувати відкритим ключем *Користувача 2*.

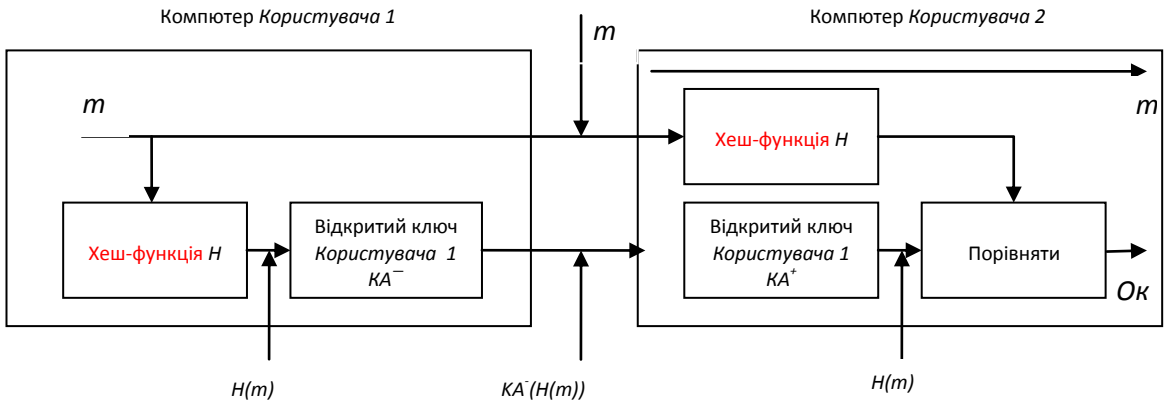


Рис. 3.6. Цифровий підпис повідомлення і використання дайджесту повідомлення

Одержавши повідомлення і зашифрований дайджест, *Користувач 2* повинен розшифрувати дайджест відкритим ключем і незалежно від *Користувача 1* обчислити дайджест повідомлення. Якщо обчислений для отриманого повідомлення дайджест і розшифрований дайджест *Користувача 1* збігаються, то можна вважати, що повідомлення було підписано *Користувачем 1*.

3.3.4. Сеансові ключі

Організуючи захищені канали після завершення фази аутентифікації з метою конфіденційності сторони зазвичай використовують унікальний загальний сеансовий ключ, який по закінченні застосування каналу анулюється. Для конфіденційності можна обмежитися тим самим ключем, що і під час встановлення з'єднання, однак сеансові ключі мають багато важливих переваг.

У разі частого використання ключа його простіше визначити третій особі. У цьому сенсі криптографічні ключі мають періодично змінюватися. Якщо зловмисник зможе перехопити великий обсяг даних, зашифрованих тим самим ключем, він зможе планувати атаки таким чином, щоб виявити важливі характеристики ключа і, можливо, одержати простий текст або навіть сам ключ. З цієї причини бажано

якомога рідше задіювати ключ аутентифікації. Крім того, подібні ключі часто змінюються за допомогою яких-небудь повільних зовнішніх механізмів, наприклад регулярних поштових відправлень або телефонограм. Такі способи обміну ключами слід звести до мінімуму.

Важливою причиною генерації окремого ключа на кожен захищений канал є гарантований захист від атак спотворення повідомлень. Використовуючи унікальні сеансові ключі під час кожного відкриття захищеного каналу, сторони, що спілкуються, застраховані від спотворення всього сеансу. Щоб захиститися від спотворення окремих повідомлень з попереднього сеансу, зазвичай потрібні додаткові заходи, зокрема додавання до повідомлень відміток часу або послідовних номерів.

Ключі аутентифікації часто організовані так, що їх заміна є досить складною операцією. З огляду на це комбінація з подібних ключів тривалої дії і короткострокових сеансових ключів – нерідко краще для реалізації захищених каналів обміну даними.

Основні положення закону України «Про електронний цифровий підпис». В Україні набув чинності ще 2003 року закон України «Про електронний цифровий підпис», проте для введення його в дію потрібні значні капітальні затрати: створення баз даних, їх постійне редагування для фізичних і юридичних осіб, перевірка значної кількості документів під час прийому заявок тощо [40].

Цей закон визначає правовий статус електронного цифрового підпису та регулює відносини, які виникають у процесі використання електронного цифрового підпису. Дія цього закону не поширюється на відносини, що виникають під час використання інших видів електронного підпису, зокрема переведеного у цифрову форму зображення власноручного підпису. Якщо міжнародним договором, згоду на обов'язковість якого надала Верховна Рада України,

встановлено дещо інші правила, ніж передбачені цим законом, то застосовуються правила міжнародного договору.

3.4. Захищена групова взаємодія

Конфіденційна групова взаємодія. Розглянемо проблему захисту взаємодії у групі з *N* користувачів від підслуховування. Найпростішою схемою забезпечення конфіденційності буде спільне використання всією групою одного секретного ключа для шифрування і розшифрування повідомлень, переданих членами групи один одному. Оскільки секретний ключ за цією схемою спільно використовують усі члени групи, необхідно, щоб вони вірили, що ключ насправді секретний. Ця вимога робить використання одного загального секретного ключа у разі групової взаємодії більш уразливим для атак порівняно з двостороннім захищеним каналом.

Альтернативним рішенням проблеми захисту взаємодії у групі є випадок, коли кожна пара членів групи спільно використовує окремі ключі. Як тільки один член групи почне втрачати інформацію, решта членів групи припиняють відправлення йому повідомлень, але продовжують застосовувати для зв'язку один з одним старі ключі. Однак, на відміну від використання найпростішої схеми забезпечення конфіденційності з підтримкою одного ключа, тепер потрібно підтримувати *N(N- l)/2* ключів, що може бути непросто.

Для запобігання підвищенню обчислювальної складності у разі групової взаємодії використовують криптосистему з відкритим ключем, за якої кожен член групи матиме власну пару (відкритий і закритий ключі), у якій відкритий ключ використовуватимуть усі члени групи для надсилання конфіденційних повідомлень. Для цього потрібно лише *N* пар ключів. Якщо одному з членів групи починають не довіряти, то його видаляють із групи, не змінюючи інші ключі.

Захищена реплікація серверів. Сервери можуть реплікуватися з різних причин, зокрема для захисту від збоїв або для підвищення продуктивності, але в будь-якому разі клієнт має отримувати достовірну відповідь.

Способом захисту клієнта від атак з метою злому системи захисту серверів є збирання відповідей усіх серверів з ідентифікацією кожного з них. Якщо відповіді від тих серверів, які не зламано (тобто аутентифікованих), становлять більшість, то клієнт може вважати, що відповідь коректна. Але подібний підхід порушує принцип прозорості процесу реплікації серверів.

Використовують рішення для захищеного реплікованого сервера, що дозволяє зберегти прозорість реплікації, перевага якого полягає в тому, що клієнти не знають про реальні репліки, тому значно простіше додавати і видаляти репліки поза їх увагою.

Сутність захисту з одночасною прозорою реплікацією серверів полягає в використанні принципу поділу секрету (*secret sharing*), за якого жоден з декількох користувачів (або процесів) не знає секрету цілком, тобто секрет може бути відкритий лише за умови їх спільної роботи. Такі схеми зменшують вірогідність несанкціонованого доступу до системи.

Приклад. Розглянемо переказ значної суми грошей з одного рахунку на інший транзакцією, що вимагає підтвердження як мінімум двома особами, кожна з яких має власний закритий ключ, який має використовуватися в парі з другим, запуск транзакції у разі використання єдиного ключа не відбудеться.

Якщо репліковані сервери захищені, то під час пошуку відповіді k з N серверів можуть дати невірну відповідь, а з цих k максимум $c \leq k$ серверів можливо дійсно зламані зловмисником, тобто у такому разі припускають, що служба має бути стійкою до k помилок, при чому c серверів, що працюють помилково, зламані навмисно.

Тепер розглянемо ситуацію, коли сервери активно реплікуються. Запит розсилається всім серверам одночасно, після чого обробляється

кожним з них, тобто кожен сервер генерує відповідь, що повертається клієнтові. У разі захищеної реплікованої групи серверів передбачено, що кожен сервер супроводжує свою відповідь цифровим підписом. Якщо r_i – відповідь сервера S_i, то $md(r_i)$ позначатиме дайджест повідомлення, обчислений сервером S_i і підписаний закритим ключем K_i^- сервера S_i.

У разі, якщо необхідно захистити клієнта від помилок, які спричинені максимальною кількістю зламаних серверів, то група серверів має приховати наслідки злому цих серверів, залишаючись у стані генерації відповіді, якій клієнт зможе довіряти. Для цього підпис окремих серверів комбінують таким чином, щоб побудова правильного підпису для результату вимагала мінімум $c+1$ підписів, тобто під час створення секретного правильного підпису реплікованими серверами зі зламаних серверів не можна зібрати цей підпис без допомоги мінімум одного «чесного» сервера.

Як приклад розглянемо групу з п'яти реплікованих серверів, які повинні мати здатність витримати злом двох із них і давати при цьому результат, якому зможе довіряти клієнт. Кожен сервер S_i надсилає відповідь r_i разом зі своїм підписом $sig(S_i, r_i) = K_i^-(md(r_i))$ клієнтові, який, відповідно, послідовно одержує п'ять триплетів $<r_i, md(r_i), sig(S_i,r_i)>$, з яких він повинен виокремити правильні. Цю ситуацію ілюструє рис. 3.7.

Кожен дайджест $md(r_i)$ обчислюється також і на клієнті. Якщо відповідь r_i, неправильна, це факт зазвичай визначають обчисленням $K_i^+(K_i^-(md(r_i)))$, однак, оскільки не можна довіряти жодному серверу окремо, цей метод не застосовують. Замість нього клієнт використовує спеціальну відкриту функцію розшифрування D, що як вихідні дані одержує набір із трьох сигнатур $V=\{sig(S, r), sig(S', r'), sig(S'', r'')\}$, а як результат генерує єдиний дайджест:

$$dout = D(V) = D(sig(S, r), sig(S', r'), sig(S'', r'')).$$

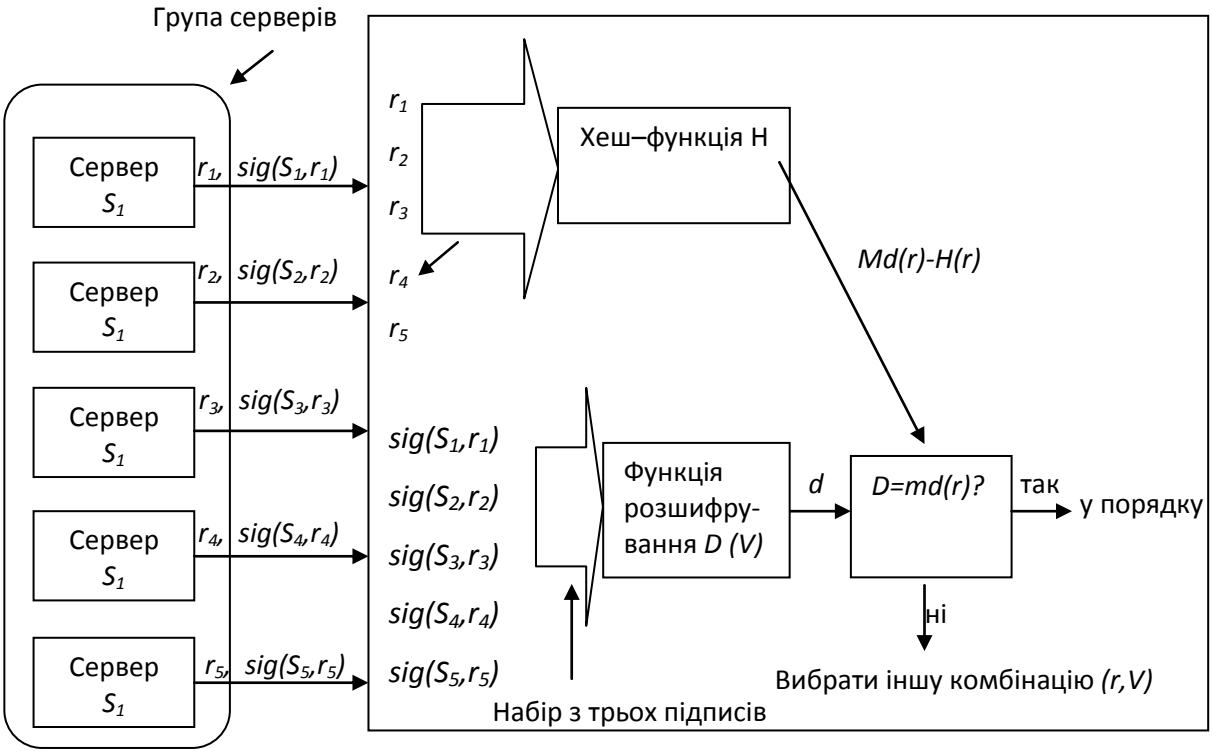


Рис. 3.7. Поділюваний секретний підпис групи реплікованих серверів

Існує 5!/(3!2!)=10 комбінацій трьох підписів, які клієнт може одержати як результат *D*. Якщо одна із цих комбінацій дасть правильний дайджест $md(r_i)$ для деякого результату r_i, то відповідь r_i клієнтом має вважатися правильною, тобто, що цей результат надали разом як мінімум три правильно працюючі сервери.

Для підвищення прозорості реплікації потрібно, щоб кожен сервер S_i розсилав свій результат r_i іншим серверам разом з відповідним підписом $sig(S_i, r_i)$. Коли сервер одержить як мінімум $c+1$ подібних повідомлень, включаючи його власне, він зможе обчислити правильний підпис для однієї з відповідей. Якщо, наприклад, це обчислення для відповіді *r* і набору *V* із $c+1$ підписів буде успішним, то він пересилає *r* і *V* клієнтові одним повідомленням. Клієнт після цього аналізує коректність *r* за рахунок перевірки підпису, тобто $md(r) = D(V)$.

Описаний алгоритм відомий як *m*, *n*- гранична схема (*m,n-threshold scheme*), де $m = c+1$, а $n = N$ (кількість серверів). У *m, n* - граничній схемі повідомлення ділиться на *n* частин, відомих за назвою «тіні» (*shadows*). Для відтворення вихідного повідомлення можуть бути

використані будь-які *m* тіней, але будь-яких *m-1* тіней для цього буде недостатньо.

3.5. Контроль доступу. Загальні питання контролю доступу

У моделі клієнт-сервер після встановлення між клієнтом і сервером захищеного каналу клієнт створював запити, які передавалися серверу та містили операції над ресурсами, контрольованими сервером. Як правило, сервер об'єктів керує декількома об'єктами. Запит клієнта зазвичай містить звертання до методу конкретного об'єкта та може бути виконаний лише за умови, що клієнт має достатні для цього права доступу (*access rights*).

Формальне підтвердження прав доступу називають контролем доступу (*access control*), у той час як видачу прав доступу - авторизацією (*authorization*). Ці два поняття тісно взаємопов'язані, і нерідко одне з них використовують замість другого.

Існує дуже багато методів контролю доступу. Один з особливо значущих методів контролю доступу до ресурсів – це створення брандмауера, що захищає прикладне програмне забезпечення або навіть усю мережу цілком. У зв'язку з підвищенням мобільності коду контроль доступу стає неможливим здійснювати лише традиційними методами, для цього розроблено ряд нових спеціалізованих технологій.

Загальні питання контролю доступу. Щоб розібратися в різних аспектах контролю доступу, розглянемо просту модель, подану на рис. 3.8, яка складається із суб'єктів (*subjects*), що надсилають запити на доступ до об'єкта (*object*), який має стани, що інкапсулюють його певні внутрішні стани і реалізації операцій об’єкта усередині кожного стану. Операції об'єкта, запити на виконання яких надсилають суб'єкти, доступні через інтерфейси. Суб'єкти виконують за завданням

користувачів процеси, але можуть також бути й об'єктами, які для здійснення своєї роботи потребують допомоги інших об'єктів.

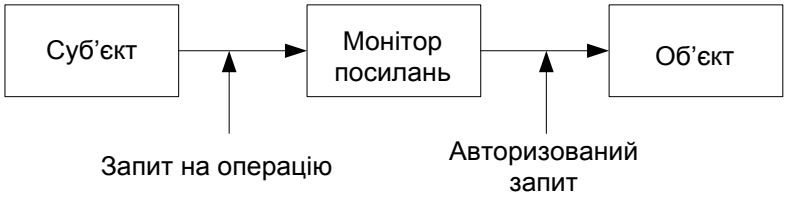


Рис. 3.8. Узагальнена модель контролю доступу до об'єктів

Керування доступом до об'єктів – це засоби, які захищають об'єкти від звертання суб'єктів, що не мають права на виклик відповідних методів, а можливо, й жодних методів. Окрім того, захист може стосуватися деяких аспектів керування об'єктами, зокрема створення, перейменування або видалення об'єктів.

Захист часто реалізується за допомогою програми, яку називають **монітор посилань** (*reference monitor*), який записує, що може робити певний суб'єкт, і вирішує, чи припустимо для цього суб'єкта виконання конкретної операції. Цей монітор викликається (наприклад, базовою довіреною операційною системою) у разі будь-яких звертань до об'єкта. Таким чином, дуже важливо, щоб монітор посилань сам по собі мав підвищений захист від атак, тобто зловмисник не зміг його обманути.

Загальноприйнятий підхід до моделювання прав доступу суб'єкта стосовно об'єктів полягає в побудові **матриці контролю доступу** (*access control matrix*), кожен суб'єкт якої подано рядком цієї матриці, кожен об'єкт – стовпцем. Якщо ввести для матриці позначення *M*, то елемент *M(s,o)* точно визначає, виконання яких операцій суб'єкт s може очікувати від об'єкта *o*. У разі звернення суб'єкта s до методу *m* об'єкта *o*, монітор посилань має перевірити, чи зазначений метод *m* у *M(s,o)*. Якщо *m* у *M(s,o)* не має, то виклику не відбудеться.

Системі може знадобитися підтримувати тисячі користувачів і мільйони об'єктів, які потребують захисту, тоді реалізація матриці

контролю доступу у вигляді матриці не є найкращим рішенням. Більшість елементів матриці будуть порожніми, оскільки один суб'єкт зазвичай має доступ до порівняно невеликої кількості об'єктів, тому для реалізації матриці контролю доступу використовуються більш ефективні способи.

Один зі способів, який найчастіше застосовується, полягає в тому, що кожний об'єкт підтримує список прав доступу суб'єктів, які бажають одержати доступ до цього об'єкта. Це реалізується за допомогою матриці контролю доступу, яка розбивається на стовпці, кожний з яких відповідає певному об'єкту, при цьому порожні елементи відкидаються. Такий спосіб реалізації називають списком контролю доступу (*Access Control List, ACL*), причому кожний об'єкт має власний, асоційований тільки з ним список ACL.

Другий підхід полягає в тому, що матриця розбивається по рядках і кожний суб'єкт одержує список мандатів (*capabilities*) для кожного з об'єктів. Мандат відповідає елементу в матриці контролю доступу. Якщо у конкретного об'єкта немає мандата, то це означає, що суб'єкт не має права на доступ до цього об'єкта.

Мандат можна порівняти з талоном – той, у кого він є, має права, зазначені в цьому талоні. Талон має бути захищений від спроб власника модифікувати його. Один зі способів захисту талона від модифікації, використовуваний у розподілених системах, полягає в захисті мандатів (списку мандатів) за допомогою підпису.

Відмінність між використанням для захисту об'єктів списків контролю доступу і списків мандатів подано на рис. 3.9. У разі використання списків контролю доступу клієнт надсилає запит на сервер, монітор посилань з'ясовує, що йому відомо про цього клієнта, і якщо клієнтові дозволяється виконувати потрібну операцію, вона буде виконана, що і показано на рис. 3.9, *а*.

Для захисту об’єктів за допомогою списків мандатів клієнт передає свій запит разом зі списком мандатів на сервер, який не зберігає відомостей про клієнта, оскільки список мандатів надає йому всю необхідну інформацію. Таким чином, сервер повинен просто перевірити, чи є в запиті коректний список мандатів і чи зазначена запитувана операція в цьому списку (рис. 3.9, *б*).

Списки ACL і мандати допомагають успішно реалізувати матрицю контролю доступу без порожніх елементів. Проте якщо не вживати ніяких додаткових заходів, і списки контролю доступу, і списки мандатів зрештою можуть стати занадто великими.

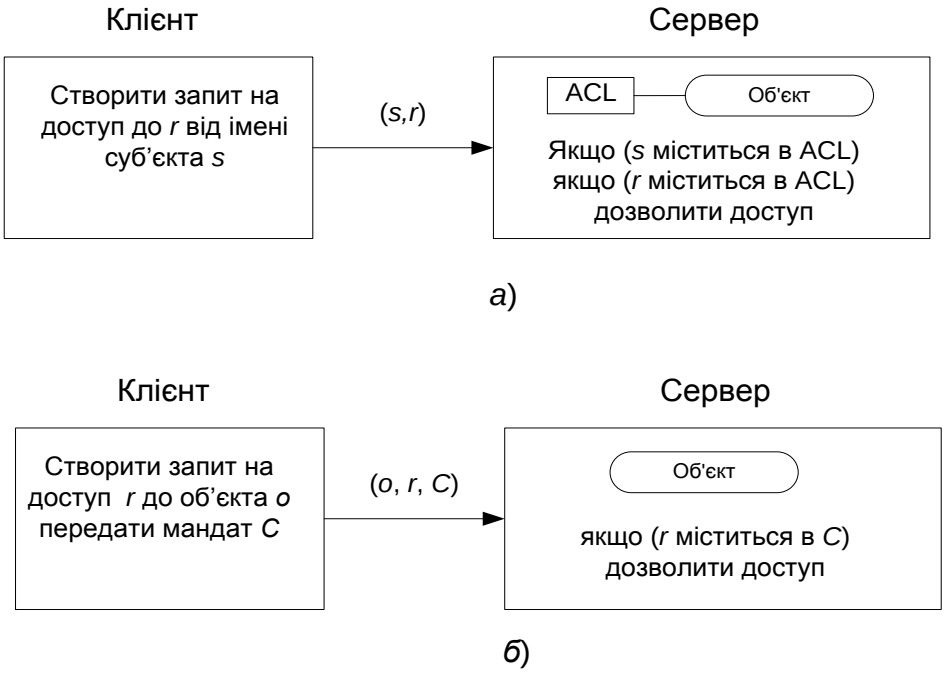


Рис. 3.9. Порівняння варіантів захисту об'єктів за допомогою ACL (*а*) і мандатів (*б*)

Один з основних способів зменшити розмір списку ACL – використання захищених доменів. **Захищений домен** (*protection domain*) – це набір пар (об'єкт, права доступу). Кожна пара ідентифікує операції, які може виконувати певний об'єкт. Запити на виконання операції завжди дозволяються усередині домену. Коли суб'єкт запитує в об'єкта виконання операції, монітор посилань спочатку знаходить захищений домен, пов'язаний з цим запитом, потім, обравши домен, він

перевіряє, чи може бути виконаний запит. Існують різні підходи щодо використання захищених доменов.

Один з підходів грунтується на побудові груп (*groups*) користувачів. Розглянемо, наприклад, web-сторінку внутрішньої мережі компанії, до якої повинні мати доступ лише співробітники компанії. Замість того щоб додавати в ACL по елементу на кожного співробітника, можна створити окрему групу, яка має список усіх співробітників. Щоразу під час звернення користувача до web-сторінки монітор посилань повинен буде тільки перевірити, чи є користувач співробітником. Користувачі, які утворюють групу співробітників, мають входити до окремого списку, захищеного від неавторизованого доступу.

3.6. Брандмауери

Складною задачею є організація захисту, що поєднує криптографічні методи з реалізацією матриць контролю доступу, через те, що всі ці способи добре працюють доти, поки сторони, які зв'язуються, використовують єдині правила взаємодії. Однак змусити дотримуватися цих правил можна, тільки якщо розробляють розподілену систему, ізольовану від інших. Значно складніше забезпечити захищеність системи, коли доступ до ресурсів, якими керує розподілена система, відкривається стороннім користувачам, наприклад, під час відправлення пошти, завантаження файлів або вивантаження форм.

Доступ ззовні до кожної з частин розподіленої системи відслідковується монітором посилань особливого типу, який називають *брандмауер* (*firewall*). Брандмауер відокремлює частину розподіленої системи від навколишнього світу, реалізація чого показана на рис. 3.10. Усі вихідні та вхідні пакети проходять через спеціальний комп'ютер і

перед тим, як будуть передані далі, перевіряються. Неавторизований трафік далі не передається. Слід зазначити, що сам брандмауер має бути надійно захищеним від будь-яких загроз захисту, тобто ніколи не виходити з ладу.

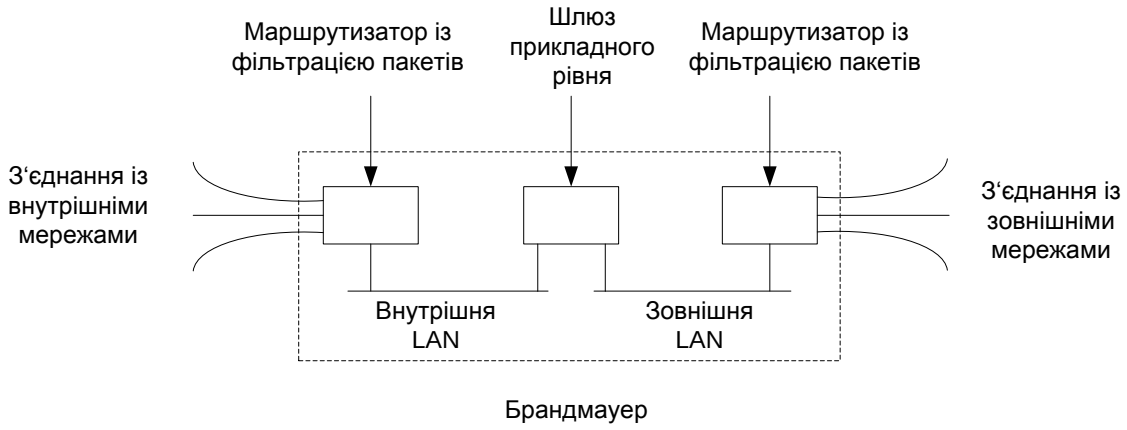


Рис. 3.10. Узагальнена реалізація брандмауера

Розрізняють брандмауери двох типів, але часто брандмауери, які використовують на практиці, становлять їх комбінацію. Найважливіший тип брандмауера – це **шлюз фільтрації пакетів** (*packet-filtering gateway*), який працює як маршрутизатор, тобто приймає рішення про те, чи пропускати мережний пакет на підставі адрес відправника й одержувача в заголовку пакета. Зазвичай (рис. 3.10) шлюз фільтрації пакетів, який перебуває поза локальною мережею (*LAN*), захищає систему від вхідних пакетів, а такий самий шлюз, що перебуває у внутрішній мережі, фільтрує вихідні пакети. Наприклад, для захисту внутрішнього web-сервера від запитів хостів, що не входять у внутрішню мережу, шлюз фільтрації пакетів має відкидати всі вхідні пакети, адресовані web-серверу.

Більш складною є ситуація, коли мережа компанії складається з декількох локальних мереж, з'єднаних між собою. Кожна внутрішня мережа може бути захищена шлюзом фільтрації пакетів, який зконфігуровано на пропускання вхідного трафіка тільки від хостів інших внутрішніх мереж. Таким чином, будуються приватні віртуальні мережі.

Другий тип брандмауерів – **шлюз прикладного рівня** (*application-level gateway*), який, на відміну від шлюзу фільтрації пакетів, що перевіряє тільки заголовки пакетів, переглядає вміст вхідних і вихідних повідомлень. Типовим прикладом може бути *шлюз пошти*, який відкидає вхідні або вихідні листи, що перевищують установлений розмір. Існують і більш складні поштові шлюзи, здатні, наприклад, відфільтрувати електронну пошту рекламного характеру (*спам*).

Другим прикладом шлюзу прикладного рівня може бути **шлюз зовнішнього доступу** до сервера цифрової бібліотеки, що дозволяє ознайомитися тільки з рефератами документів. Якщо зовнішній користувач хоче одержати щось ще, то починає працювати протокол електронних платежів. Внутрішній користувач має прямий доступ до бібліотечних служб.

Існує особливий тип шлюзу прикладного рівня, який називають *прокси-шлюзом* (*proxy gateway*). Цей тип брандмауера приховує роботу деяких типів прикладних програм і гарантує проходження через нього лише повідомлень, що задовольняють заданим критеріям.

3.7. Керування захистом в розподілених системах

3.7.1. Керування ключами

Створення сеансових ключів. Коли *Користувач 1* хоче організувати захищений канал зв'язку з *Користувачем 2*, він може використати для ініціювання зв'язку відкритий ключ *Користувача 2*. Якщо *Користувач 2* погоджується почати взаємодію, то він повинен створити сеансовий ключ і повернути його *Користувачу 1*, зашифрувавши відкритим ключем *Користувача 1*. Після шифрування загального сеансового ключа його можна передавати мережею.

Подібна схема може використовуватися для генерації та поширення сеансового ключа за умови, що *Користувач 1* і *Користувач 2* уже мають загальний секретний ключ. Однак необхідно, щоб сторони, між якими встановлюється зв'язок, могли організувати захищений канал, тобто у них уже має бути якийсь спосіб створення і поширення ключів. Таким же чином здійснюється взаємодія у тому разі, коли загальний секретний ключ створюється довіреною третьою стороною, наприклад центром розповсюдження ключів (*Key Distribution Center - KDC*).

Спосіб створення загального ключа і передачі його небезпечними каналами, який зазвичай використовується, – це обмін ключами за Диффі-Хеллманом (*Diffie-Hellman key exchange*). Цей протокол працює в такий спосіб. Припустімо, *Користувач 1* і *Користувач 2* бажають створити загальний секретний ключ. Спочатку вони повинні домовитися про використання двох більших чисел (*n* і *g*), які задовольняють деяким математичним обмеженням і їх немає потреби приховувати від сторонніх. Далі *Користувач 1* обирає велике випадкове число, скажімо, *x*, яке він тримає в секреті. *Користувач 2* також обирає своє секретне число, назвемо його *y*. У цей момент, як видно з рис. 3.11, є вся інформація, необхідна для створення секретного ключа.

Користувач 1 надсилає *Користувачеві 2* число $g^x \bmod n$, разом з *n* і *g*. Важливо відзначити, що цю інформацію можна відправляти відкритим текстом, тому що, маючи значення $g^x \bmod n$, обчислити *x* дуже важко. Одержавши повідомлення, *Користувач 2* обчислює $(g^x \bmod n)^y$, що дорівнює $g^{xy} \bmod n$. Окрім того, він надсилає *Користувачу 1* число $g^y \bmod n$, а він, у свою чергу, обчислює $(g^y \bmod n)^x = g^{xy} \bmod n$. Відповідно, *Користувач 1* і *Користувач 2*, і тільки вони, мають тепер загальний секретний ключ $g^{xy} \bmod n$. Відзначимо, що

нікому з них не знадобилося повідомляти іншому своє секретне число (відповідно, *x* або *y*).

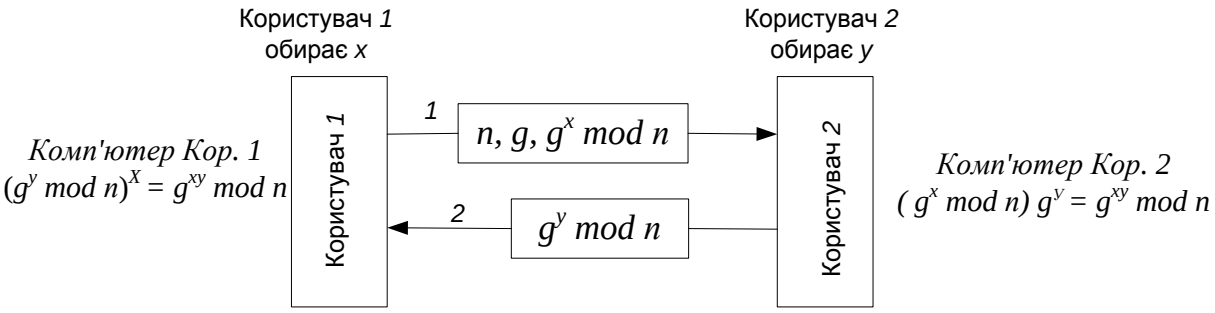
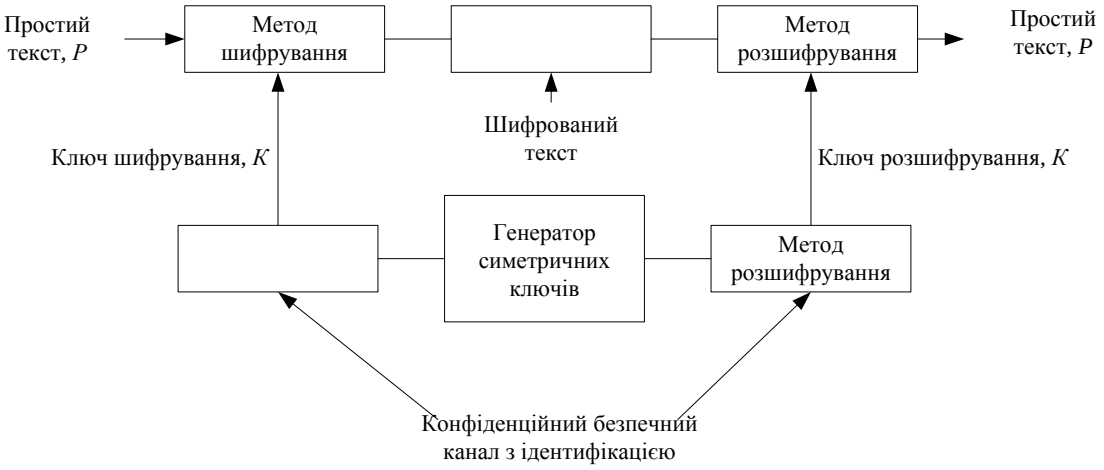


Рис. 3.11. Обмін ключами за Диффі-Хеллманом

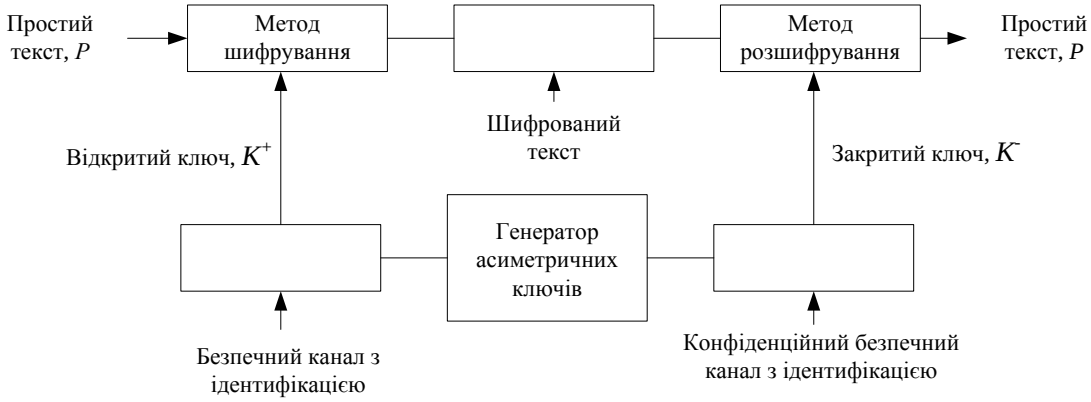
Протокол Диффі-Хеллмана можна розглядати як криптосистему з відкритим ключем, тому що для *Користувача 1*, *x* – це його закритий ключ, а $g^{xy} \bmod n$ – відкритий. Процедура захищеного поширення відкритих ключів аналогічна протоколу Диффі-Хеллмана.

Одна з найбільш складних частин керування ключами – це **поширення вихідних ключів**. У симетричних криптосистемах вихідний загальний секретний ключ має поширюватися безпечними каналами з аутентифікацією і конфіденційно (рис. 3.12, *а*). Якщо у *Користувача 1* і *Користувача 2* немає ключів для організації захищеного каналу, то необхідно обмінятися ключами іншим способом, тобто вони мають увійти в контакт не через мережу, наприклад, зв'язатися по телефону або надіслати ключ на дискеті бандероллю.

У разі використання криптосистеми з відкритим ключем необхідно поширити відкритий ключ так, щоб одержувачі були впевнені, що цей ключ дійсно є парою заявленому закритому. Хоча відкритий ключ може бути відправленим навіть простим текстом, необхідно, щоб канал, яким він пересилається, забезпечував аутентифікацію. Закритий ключ необхідно пересилати тільки безпечним каналом, з ідентифікацією і конфіденційно.



а)



б)

Рис. 3.12. Поширення секретного (*а*) і відкритого ключа (*б*)

Аутентифіковане поширення відкритого ключа на практиці здійснюється за допомогою сертифікатів відкритого ключа (*public key certificates*), які складаються з відкритого ключа і рядка, що визначає сутність, з якою асоційований цей ключ, зокрема користувача, хост або якийсь спеціальний пристрій. Відкритий ключ і ідентифікатор разом підписує організація, яка сертифікує, (*certification authority*), і цей підпис додається до сертифіката (найменування такої організації є обов'язковою складовою сертифіката). Підпис виконується закритим ключем K^-_{CA}, який належить організації, що сертифікує. Відповідний закритому ключу відкритий ключ K^+_{CA} не вважається секретним. Так, відкриті ключі різних організацій, що сертифікують, убудовані в більшість web-браузерів і поширюються разом із файлами програм.

Сертифікати відкритого ключа використовуються в такий спосіб. Припустімо, клієнт бажає встановити, чи належить насправді відкритий ключ, що міститься в сертифікаті, зазначеній сутності. Він за допомогою відкритого ключа відповідної організації, яка сертифікує, перевіряє підпис сертифіката. Якщо підпис сертифіката відповідає парі (відкритий ключ та ідентифікатор), то клієнт уважає, що відкритий ключ дійсно належить зазначеній сутності.

Вважаючи сертифікат придатним, клієнт насправді вірить, що сертифікат не підроблений, зокрема, клієнт припускає, що відкритий ключ K^{+}_{CA} дійсно належить відповідній організації, яка сертифікує. Якщо є сумніви, то можна перевірити правильність K^{+}_{CA} через інший сертифікат, отриманий від іншої організації, якій, можливо, користувач довіряє більше.

Такі ієрархічні *моделі довіри* (*trust models*), за якими організаціям, які сертифікують, найвищого рівня довіряють усі фактично завжди застосовують на практиці. Наприклад, конфіденційна пошта (*Privacy Enhanced Mail, PEM*) використовує трирівневу модель довіри, за якою організації нижнього рівня ідентифікують організації сертифікації правил (*Policy Certification Authorities, PCA*), які, у свою чергу, ідентифікує організація реєстрації правил в Internet (*Internet Policy Registration Authority, IPRA*). Якщо користувач не довіряє *IPRA* або не вважає, що може безпечно спілкуватися з *IPRA*, то ніколи не повірить листам, переданим поштою *PEM*.

Кожний сертифікат видається не назавжди, а має певний *строк життя* (*lifetime*). У разі, коли організація, яка сертифікує, видає постійні сертифікати, то такий сертифікат означає, що відкритий ключ для сутності, яку він ідентифікував, завжди є коректним. Якщо закритий ключ сутності, яку ідентифікують, один раз буде скомпрометований, то клієнти, які нічого не підозрюють, не зможуть використати відкриті ключі (на відміну від клієнтів, які є

зловмисниками). Для подібних випадків потрібний механізм *відкликання* (*revoke*) сертифіката, який дозволив би зробити відомим той факт, що сертифікат більше недійсний.

Існує кілька способів відкликання сертифіката. Один із традиційних методів – скористатися списком відкликаних сертифікатів (*Certificate Revocation List, CRL*), який регулярно публікує організація, що сертифікує. Коли клієнт перевіряє сертифікат, він переглядає CRL, щоб визначити, чи не було цей сертифікат відкликано. Це означає, що клієнт повинен зв'язуватися з організацією, яка сертифікує щоразу з виходом нового списку CRL. Відзначимо, що якщо CRL публікується щодня, то і на відкликання сертифікатів дається лише один день. Таким чином, скомпрометований сертифікат зловмисник може використати доти, поки він не буде включений у черговий список CRL. Відповідно, час між публікаціями CRL не має бути занадто великим, але треба враховувати, що одержання CRL вимагає додаткових витрат.

Альтернативний підхід передбачає обмеження строку життя сертифіката, тобто дія сертифікатів після закінчення визначеного часу автоматично припиняється. Якщо з якихось причин сертифікат був відкликаний до припинення часу його дії, то організація, яка сертифікує, може опублікувати відомості про це в CRL. Однак подібний підхід спонукає клієнтів переглядати останній список CRL незалежно від часу одержання сертифіката і вони змушені зв'язуватися з організацією, яка сертифікує, або довіреною базою даних, що містить останній список CRL.

Якщо скоротити строк життя сертифікатів майже до нуля, то це означатиме повну відмову від використання сертифікатів. Замість цього клієнт завжди повинен зв'язуватися з організацією, яка сертифікує, і перевіряти придатність відкритого ключа, тобто організація має постійно бути на зв'язку.

На практиці сертифікати мають обмежений строк життя. Для прикладного програмного забезпечення, яке використовує Інтернет-технології, час життя часто становить не більше одного року. Такий підхід вимагає регулярної публікації CRL і перевірки терміну дії сертифіката. Практичний досвід показує, що прикладне програмне забезпечення клієнтів рідко звіряється з CRL, автоматично вважаючи сертифікати придатними до закінчення їх терміну дії.

3.7.2. Електронні платіжні системи

Оплата завжди передбачає наявність двох учасників: продавця і покупця. Здебільшого до процесу залучаються також один або два банки, тому що покупець має бути впевнений, що продавець одержить свої гроші.

Існує кілька способів організації платіжних систем. Найчастіше в реальному житті використовуються готівка, чеки і кредитні картки. У системі, в якій розрахунки здійснюються готівкою, покупці повинні одержати у своєму банку готівку, яку потім передають продавцеві в обмін на товари. Продавці згодом можуть використати ці гроші, щоб заплатити кому-небудь ще або покласти їх на рахунок у своєму банку.

Інакше працює система на основі чеків. У цьому разі покупці мають завірені підписом документи свого банку у вигляді чеків, які дають право кожному, хто надав ці чеки в зазначений банк, на переказ коштів з рахунку покупця. На практиці банки покупця і продавця після пред'явлення продавцем чека у свій банк самі здійснюють взаємні розрахунки.

Третій спосіб заснований на використанні кредитних карток. Для оплати в системі за кредитними картками покупець вимагає від банку переказу коштів на рахунок продавця, надаючи йому кредитну картку і дозволяючи її зчитування. Реальне перерахування коштів виконується,

коли продавець надає результат зчитування у свій банк, що зв'язується з банком покупця.

Існують також інші форми оплати. Наприклад, часто можна заплатити за товар, перерахувавши кошти з банку покупця в банк продавця авансом. Одержавши інформацію про перерахування коштів а свій рахунок, продавець здійснює доставку товарів. Цю схему платежу зазвичай використовують, якщо компанія-продавець не знає покупця і не хоче постачати продукцію до одержання грошей.

У разі використання авізо, покупець дозволяє продавцеві списувати визначені суми з його банківського рахунку. Такий підхід часто використовують, якщо потрібно здійснювати регулярні платежі, зокрема членські внески або щомісячні комунальні платежі за газ і електрику.

3.7.3. Захист в електронних платіжних системах

Захист відіграє в електронних платіжних системах важливу роль, вимагаючи авторизованого доступу до власного банківського рахунка, при чому захист у платіжних системах відіграє значно важливішу роль, ніж в інших типах розподілених систем.

У системах, яка здійснює розрахунки готівкою, найбільш поширені автоматичні касові машини (*Automatic Teller Machines, ATM*), або банкомати, які контактують з однією з банківських баз даних. Коли *Користувач 1* хоче одержати гроші у своєму банку, він повинен спочатку аутентифікувати себе, тобто використати спеціальний маркер у формі картки з магнітною смугою або чіпом. Ця картка, яку зчитує банкомат, захищена персональним ідентифікаційним номером (*Personal Identification Number, PIN*). *Користувач 1* повинен набрати свій номер PIN, після чого ATM зв'язується з банком. Якщо

аутентифікація виявилася успішною, то *Користувачу 1* дозволено одержати визначену суму грошей.

Існує й інший підхід. *Користувач 1* може, не вдаючись до допомоги банкомата, використати для переказу коштів зі свого рахунка домашній персональний комп'ютер.

Один зі способів, застосовуваних в електронних системах, – використання смарт-карти, яка може зберігати цифрові гроші (*digital money*), що існують у формі бітів. *Користувач 1* може вказати, що бажає зберігати свої цифрові гроші локально, на власному жорсткому диску. Використання цифрових грошей у схемах, розроблених для готівки, вимагає надійного захисту від шахрайства. Зокрема, потрібно запобігти використанню цифрових грошей більше одного разу або виготовленню фальшивих грошей. У такому разі існує нагальна потреба в механізмі забезпечення цілісності даних.

Приклад. Коли *Користувач 1* використовує свої цифрові гроші, щоб заплатити *Користувачу 2, Користувач 2* повинен мати змогу перевірити справжність грошей. Цифрові гроші *Користувач 2* вважатиме справжніми, якщо він може піти у свій банк і покласти їх на рахунок, але банк *Користувача 2* теж має впевнетись у справжності грошей, тобто банк *Користувача 2* може зв'язатися з банком, який випускав ці гроші. Після цього банк *Користувача 2* аутентифіковуватиме банк *Користувача 1*.

Розглянемо систему, яка грунтується на використанні, що як необхідну умову проходження платежу застосовує стандартні процедури аутентифікації й авторизації чека або кредитної картки, яку *Користувач 1* видав або пред'явив.

Приклад. Приймаючи оплату електронним чеком *Користувач 2* та його банк мають впевнетись, що чек захищений від підробок.

Платіжні системи з використанням чеків або кредитних карток потребують суворіших вимог до захисту ніж системи на основі готівки.

Приклад. Стандартний спосіб купівлі товарів через Internet – надсилання замовлення із зазначенням номера кредитної картки. Хоча номер кредитної картки

часто шифрується до пересилання його продавцеві, він не є еквівалентом підпису на чеку або даних зчитування у разі платежу кредитною карткою.

Оскільки порівняно легко (теоретично) заперечувати сам факт замовлення у разі використання кредитних карток, то запобігти відмові в оплаті можна, вимогаючи від покупця передавати у своїй частині транзакції цифровий підпис.

У зв'язку з можливістю відмови від оплати створено механізм, який гарантує, що транзакція виконуватиметься автоматично, тобто вона або буде виконана повністю, або не здійсниться взагалі. Ця вимога випливає з ймовірності виникнення не тільки відмови від оплати, але й не визнання факта оплати. Якщо система підтвердить *Користувачу 1*, що оплата була виконана, то це має означати, що *Користувач 2* одержав гроші й не зможе згодом відмовитися від цього.

Електронні платіжні системи передбачають також і вимогу надійності доступу. Зокрема, система в цілому повинна бути легкодоступною і мати високий ступінь надійності.

Хоча запити до вимог захисту в електронних платіжних системах вищі, ніж у традиційних розподілених системах, вони можуть бути реалізовані з використанням описаних технологій. Однак існує характерна риса, що заслуговує на особливу увагу і нерідко вирізняє електронні платіжні системи серед розподілених систем прикладного призначення – це закритість.

Захистити платіж від зловмисників порівняно просто, користуючись стандартними методами шифрування, але у такому разі важко забезпечити анонімність платіжу. У системах платежів, які здійснюють готівкою, підтримати анонімність платіжу нескладно: *Користувач* 1 просто передає *Користувачу 2* гроші в обмін на придбані ним товари. *Користувачу 1* зазвичай немає потреби знати, хто такий *Користувач 2*, і навпаки.

Однак, якщо *Користувач 1* і *Користувач 2* використовують електронну платіжну систему, то суттєве значення має захист мікроданих (*microdata*), якими є елементи даних, що супроводжують кожну транзакцію. Якщо зібрати їх разом, вони здатні відкрити одну зі сторін, залучених у цю транзакцію.

Приклад. Розглянемо анонімний платіж, у процесі якого записується ім'я хоста, з якого ініційовано цей платіж, і час оплати. Порівняння цієї інформації з записами про входи в систему дозволить відразу ж визначити особистість покупця. Інша інформація, доступ до якої зазвичай здійснюється аналогічно, – це певні значення атрибутів, які дозволяють одержати повний профіль інтересів покупця.

На практиці захист системи від порушення закритості полягає в тому, щоб зробити покупця анонімним, хоча у цьому як правило немає потреби. **Анонімність** у такому разі означає неможливість аутентифікації покупця у процесі або по закінченні транзакції. Іноді необхідно пізнати особистість, наприклад, під час дискусії. Таку ситуацію називають *умовною анонімністю* (*conditional anonymity*).

Є й інший підхід, який полягає у прихованні особистості покупця під псевдонімом. **Псевдонім** – це альтернативне ім'я, використовуване під час виконання транзакції або серії транзакцій, яке має специфічну властивість: користувача фактично можна ідентифікувати за псевдонімом, але довідатися дійсні дані користувача за псевдонімом покупця неможливо. Зазвичай псевдоніми використовуються для реалізації умовної анонімності.

Щоб зрозуміти, як приховати від платіжної системи зайві дані, розглянемо спочатку систему на основі традиційних методів платежу. Кожен стовпець у табл. 3.2 містить атрибут транзакції, який варто було б приховати. Розглядають п'ять таких атрибутів: особистість продавця, особистість покупця, дата транзакції, сплачена сума й інформація про оплачуваний предмет. У кожному рядку зазначено одну зі сторін, залучених у транзакцію. Виокремлюють три сторони: продавець, покупець і банк, а також відслідкувати роботу транзакції може

випадковий спостерігач (людина з черги, якщо товар придбано в магазині).

Таблиця 3.2. Приховання інформації у традиційних платіжних системах

	Продавець	**Покупець**	**Дата**	**Сума**	**Товар**
Продавець	Повністю	Частково	Повністю	Повністю	Повністю
Покупець	Повністю	Повністю	Повністю	Повністю	Повністю
Банк	Ні	Ні	Ні	Ні	Ні
Спостерігач	Повністю	Частково	Повністю	Повністю	Повністю

Для кожного учасника транзакції визначається елемент, що показує, наскільки добре він знатиме конкретний атрибут.

Приклад. Покупець у платіжній системі на основі готівки бачить продавця, дату транзакції, суму, а також придбаний товар. Однак продавець зазвичай у разі купівлі товару в магазині, не має потреби у знанні особистості покупця.

Банк як правило не знає нічого, хоча він має змогу контролювати суми платежів спостерегаючи за тим, скільки грошей продавець у результаті виконання транзакції отримав на рахунку. Сам процес виконання транзакції довгостроково не зберігають, що робить покупця повністю анонімним. Продавцеві у разі особливо великих транзакцій анонімним бути не потрібно. Більшість банків за законом зобов'язані робити запис про транзакції, у процесі яких продавець отримує гроші.

Стосовно приховання інформації, наявна ситуація, коли абсолютно всі атрибути бачитимуть усі учасники взаємодії у платіжних системах. Проте спостерігач зазнає певних труднощів, аутентифікуючи покупця, до того ж банк ніколи не отримує інформацію про те, що саме придбано покупцем, як показано в табл. 3.3.

Таблиця 3.3. Приховання інформації у традиційних системах, які використовують кредитні картки

	Продавець	**Покупець**	**Дата**	**Сума**	**Товар**
Продавець	Повністю	Повністю	Повністю	Повністю	Повністю
Покупець	Повністю	Повністю	Повністю	Повністю	Повністю
Банк	Повністю	Повністю	Повністю	Повністю	Ні
Спостерігач	Повністю	Частково	Повністю	Повністю	Повністю

Зрозуміло, що оплата кредитною карткою не забезпечує особливої закритості, залишаючи лише анонімність. Електронні версії платіжних систем мають надавати такий самий, а по можливості і більший рівень захисту. Наприклад, неприпустимо, щоб номер кредитної картки пересилався відкритими мережами у вигляді простого тексту.

3.8. Протокол Kerberos

Протокол Kerberos бувло створено в Массачусетському технологічному інституті за проектом *Athena,* однак загальнодоступним цей протокол став, починаючи з версії 4. Після того, як фахівці дослідили новий протокол, автори розробили і запропонували чергову версію – Kerberos 5, яку було прийнято як стандарт IETF. Вимоги реалізації протоколу викладено в документі RFC 1510, у специфікації RFC 1964 описується механізм і формат передавання жетонів безпеки в повідомленнях Kerberos.

Протокол Kerberos пропонує механізм взаємної аутентифікації клієнта і сервера перед установленням зв'язку між ними, при цьому в протоколі враховано той факт, що початковий обмін інформацією відбувається у незахищеному середовищі, а передані пакети можуть бути перехоплені й модифіковані. Отже, протокол є зручним для застосування в Internet й аналогічних мережах.

3.8.1. Основні концепції протоколу Kerberos

Основна концепція протоколу Kerberos дуже проста – якщо є секрет, відомий лише двом, то будь-який з його зберігачів може з легкістю впевнитися, що взаємодіє зі своїм напарником. Для цього йому достатньо перевірити, чи знає його співрозмовник загальний секрет.

Приклад. Припустімо, *Користувач 1* часто надсилає повідомлення *Користувачу 2*, який використовує їх дані лише тоді, коли абсолютно впевнений у тому, що вони надійшли саме від *Користувача 1*. Щоб ніхто не зміг підробити листи, вони домовилися між собою про пароль, який знають тільки вони вдвох. Якщо з повідомлення можна буде зробити висновок, що відправник знає пароль, то *Користувач 2* може з упевненістю сказати, що повідомлення надійшло від *Користувача 1*.

Тепер ***Користувачу 2*** та ***Користувачу 1*** залишилося тільки вирішити, яким чином показати знання пароля. Можна просто внести його в текст повідомлення, наприклад, після підпису: «***Користувач 1***, нашПароль». Це було б просто, зручно і надійно, якщо б ***Користувач 1*** і ***Користувач 2*** були впевнені, що ніхто інший не читає їхні листи. Пошта передається мережею, в якій працюють й інші користувачі, а серед них може зустрітись зловмисник, що постійно під'єднує до мережі аналізатор пакетів. Абсолютно зрозуміло, що ***Користувач 1*** не може просто вказати пароль у тілі листа. Щоб пароль залишався секретом, про нього не можна говорити відкрито, потрібно лише дати співрозмовнику знати, що пароль відомий відправнику.

Протокол Kerberos вирішує цю проблему засобами криптографії з секретним ключем. Замість того, щоб повідомляти один одному пароль, учасники сеансу зв'язку обмінюються криптографічним ключем, знання якого підтверджує особистість співрозмовника. Але щоб така технологія виявилася працездатною, необхідно, щоб загальний ключ був симетричним, тобто, він має забезпечувати як шифрування, так і дешифрування інформації. Тоді один учасник використовує його для шифрування даних, а другий за допомогою цього ключа розшифровує їх.

3.8.2. Аутентифікатори

Простий протокол аутентифікації з секретним ключем починає діяти, коли хтось запрошує дозвіл на доступ до мережі. Щоб довести

своє право на вхід, користувач пред'являє аутентифікатор (*authenticator*) у вигляді набору даних, зашифрованих секретним ключем. Отримавши аутенітифікатор, підсистема захисту розшифровує його і перевіряє отриману інформацію, щоб переконатися в успішності дешифрування. Зрозуміло, зміст набору даних треба постійно змінювати, інакше зловмисник може перехопити пакет і скористатися його змістом для входу в систему. Якщо перевірка пройшла успішно, то це означає, що відвідувачу відомий секретний код, а оскільки цей код знає тільки він і підсистема захисту, то користувач насправді той, за кого себе видає.

Може статися й так, що відвідувач захоче виконати взаємну аутентифікацію. У цьому разі використовується той самий протокол, але з деякими змінами й у зворотному порядку. Тепер підсистема захисту витягує з вихідного набору даних аутентифікатором частину інформації, а потім шифрує її, перетворюючи на новий аутентифікатор, і передає назад відвідувачеві, котрий, у свою чергу, розшифровує інформацію і порівнює її з вихідною. Якщо все збігається, то підсистема захисту знає секретний ключ.

Приклад. Припустімо, *Користувач 1* та *Користувач 2* домовилися: перед тим, як пересилати інформацію між своїми комп'ютерами, вони з допомогою відомого тільки їм двом секретного ключа перевірятимуть, хто перебуває на іншому кінці лінії.

Користувач 1 надсилає *Користувачу 2* повідомлення, що містить ім'я *Користувача 1* у відкритому вигляді й аутентифікатор, зашифрований за допомогою секретного ключа. У протоколі аутентифікації таке повідомлення є структурою даних із двома полями: перше поле містить ім'я, друге – поточний час на робочій станції *Користувача 1*.

Користувач 2 отримує повідомлення і бачить, що воно надійшло від когось, хто називає себе *Користувачем 1*. Він використовує секретний ключ і дешифрує час відправлення повідомлення. Завдання *Користувача 2* значно спрощується, якщо його комп'ютер працює синхронно з комп'ютером *Користувача 1*, тому годинники на комп'ютерах обох користувачів ніколи не розходяться більше, ніж на

5 хв. У цьому разі *Користувач 2* може порівняти розшифрований час із часом на своєму годиннику, і, якщо різниця становитиме більше 5 хв., то він відмовиться визнавати справжність аутентифікатора. Якщо ж час виявляється в межах допустимого відхилення, то можна з великою часткою впевненості припустити, що аутентифікатор прийшов саме від *Користувача 1*.

Користувач 2 шифрує час із повідомлення *Користувача 1* та вносить його у власне повідомлення, яке відправляє *Користувачу 1*.

Користувач 2 повертає *Користувачу 1* не всю інформацію з його аутентифікатором, а лише час. Якби він відправив усе одразу, то у *Користувача 1* могла б зародитися підозра, що зловнисник, вдаючи себе *Користувачем 2*, скопіював аутентифікатор з вихідного повідомлення *Користувача 1* і без змін відправив його назад.

Користувач 1 отримує відповідь *Користувача 2*, дешифрує її та порівнює отриманий час із часом, який було зазначено у вихідному аутентифікаторі. Якщо він збігається, то людина, з якою відбувається спілкування, змогла розшифрувати повідомлення *Користувача 1*, а отже, вона знає секретний ключ. А оскільки ключ знає *Користувач 1* та *Користувач 2*, то це означає, що саме *Користувач 2* отримав повідомлення *Користувача 1* і відповів на нього.

3.8.3. Керування ключами

У разі використання простих протоколів виникає проблема, як домовитись щодо місця та способу використання секретних ключів, а також про їх значення. Якщо *Користувачем 1* є клієнтська програма, встановлена на робочій станції, а *Користувачем 2* – служба на мережному сервері, то домовитися вони ніяк не можуть. Проблема ще більше ускладнюється у тих випадках, коли *Користувачу 1* - клієнту потрібно надсилати повідомлення на кілька серверів, тоді для кожного сервера йому доведеться мати окремі ключі. Та й службі на ім'я *Користувач 2* буде потрібно стільки секретних ключів, скільки у нього клієнтів. Якщо кожному клієнту для підтримання зв'язку з кожною службою потрібен індивідуальний ключ, і такий самий ключ потрібен кожній службі для кожного клієнта, то проблема обміну ключами стає

надзвичайно гострою. Необхідність збереження і захисту такої множини ключів на величезній кількості комп'ютерів створює неймовірний ризик для всієї системи безпеки.

У протоколі *Kerberos* вирішена проблема керування ключами введенням трьох учасників зв'язку, які відповідають за його безпеку: клієнт, сервер і довірений посередник між ними, роль якого відіграє *центр розподілу ключів* (*Key Distribution Center, KDC*).

Центр розподілу ключів KDC - служба, яка працює на фізично захищеному сервері, веде базу даних з інформацією про облікові записи всіх головних абонентів безпеки своєї області. Разом з інформацією про кожного абонента безпеки база даних *KDC* зберігає криптографічний ключ, відомий тільки цьому абоненту і службі *KDC*. Цей ключ, який називають довготривалим, використовується для зв'язку користувача системи безпеки із центром розподілу ключів. У разі використання протоколу *Kerberos* на практиці найчастіше довготривалі ключі генеруються на основі пароля користувача, який вказується під час входу в систему.

Коли клієнтові потрібно звернутися до сервера, він передусім надсилає запит до центру *KDC*, який у відповідь відправляє кожному учаснику майбутнього сеансу копії унікального сеансового ключа (*session key*), що діють протягом короткого часу. Призначення цих ключів – здійснення аутентифікації клієнта і сервера. Копія сеансового ключа, яка пересилається на сервер, шифрується за допомогою довгострокового ключа цього сервера, а яка надсилається клієнту - довгострокового ключа цього клієнта.

Теоретично для виконання функцій довіреного посередника центру *KDC* достатньо надіслати сеансові ключі безпосередньо абонентам безпеки. Однак на практиці реалізувати таку схему надзвичайно складно, оскільки серверу довелося б зберігати свою копію сеансового ключа в пам'яті доти, поки клієнт не зв'яжеться з ним.

Проте сервер обслуговує не одного клієнта, тому йому потрібно зберігати паролі всіх клієнтів, які можуть до нього звернутися. За таких умов керування ключами вимагає значної витрати серверних ресурсів, що обмежує масштабованість системи, а також висуває більш жорсткі вимоги до якості зв'язку, не відповідність яким може призвести до того, що запит від клієнта, який уже отримав сеансовий пароль, надійде на сервер раніше, ніж повідомлення *KDC* із цим паролем. У результаті цього серверу доведеться почекати з відповіддю доти, поки він не одержить свою копію сеансового пароля, тобто зберегти стан сервера, а це накладає на його ресурси додаткове навантаження. З огляду на це на практиці застосовується інша схема керування паролями, яка робить протокол *Kerberos* більш ефективним.

3.8.4. Сеансові мандати

У відповідь на запит клієнта, який має намір під'єднатися до сервера, служба *KDC* надсилає обидві копії сеансового ключа клієнта. Повідомлення, призначене клієнту, шифрується за допомогою довгострокового ключа, спільного для цього клієнта і *KDC*, а сеансовий ключ для сервера разом з інформацією про клієнта вкладається у блок даних, який називають сеансовим мандатом (*session ticket*). Потім сеансовий мандат цілком шифрується за допомогою довгострокового ключа, який знають лише служба *KDC* і цей сервер. Після цього вся відповідальність за обробку мандата, який містить шифрований сеансовий ключ, покладається на клієнта, який повинен повернути його на сервер.

В такому разі функції служби *KDC* обмежуються генерацією мандата, тобто їй більше не потрібно стежити за тим, чи всі надіслані повідомлення доставлені відповідним адресатам. Розшифрувати клієнтську копію сеансового ключа може лише той, хто знає секретний

довготривалий ключ цього клієнта, а щоб прочитати вміст сеансового мандата, потрібен секретний код сервера.

Отримавши відповідь *KDC*, клієнт отримує з неї мандат і свою копію сеансового ключа, які поміщає в безпечне сховище, яке міститься не на диску, а в оперативній пам'яті. Коли виникає потреба зв'язатися із сервером, клієнт надсилає йому повідомлення, що складається з мандата, який зашифрований із застосуванням довготривалого ключа цього сервера, і власного аутентифікатора, зашифрованого за допомогою сеансового ключа. Цей мандат у поєднанні з аутентифікатором становить посвідчення, за яким сервер визначає «особу» клієнта.

Сервер, отримавши «посвідчення особи» клієнта, передусім за допомогою свого таємного ключа розшифровує сеансовий мандат і витягує з нього сеансовий ключ, який потім використовує для дешифрування аутентифікатором клієнта. Якщо все виконується без збоїв, то сервер вважає, що посвідчення клієнта видано довіреним посередником, тобто службою *KDC*. Клієнт може вимагати від сервера здійснення взаємної аутентифікації, тоді сервер за допомогою своєї копії сеансового ключа шифрує мітку часу з аутентифікатором клієнта і в такому вигляді пересилає її клієнтові як власний аутентифікатор.

Одна з переваг застосування сеансових мандатів полягає в тому, що серверу не потрібно зберігати сеансові ключі для зв'язку з клієнтами, оскільки вони містяться в кеш-пам'яті посвідчень (*credentials cache*) клієнта, який надсилає мандат на сервер кожного разу, коли хоче зв'язатися з ним. Сервер, зі свого боку, отримавши від клієнта мандат, дешифрує його і витягує сеансовий ключ. Коли потреба в цьому ключі зникає, сервер може стерти його зі своєї пам'яті.

Такий метод має ще одну перевагу: у клієнта зникає потреба звертатися до центру *KDC* перед кожним сеансом зв'язку з конкретним сервером. Сеансові мандати можна використовувати багато разів, щоб

запобігти їх розкраданню встановлюється термін придатності мандата, який *KDC* зазначає у структурі даних. Цей термін визначає політика *Kerberos* для конкретного використання. Зазвичай термін придатності мандатів не перевищує восьми годин, тобто стандартної тривалості одного сеансу роботи в мережі. Коли користувач від'єднується від неї, кеш-пам'ять посвідчень обнуляється і всі сеансові мандати разом із сеансовими ключами знищуються.

3.8.5. Мандати на видання мандатів

Довготривалий ключ користувача генерується на основі його пароля.

Приклад. Коли *Користувач 1* реєструється, клієнт *Kerberos*, установлений на його робочій станції, пропускає вказаний користувачем пароль через функцію одностороннього хешування (всі реалізації протоколу *Kerberos* мають обов'язково підтримувати алгоритм *DES-CBC-MD5*, хоча можуть застосовуватися й інші алгоритми), у результаті чого генерується криптографічний ключ.

У центрі *KDC* довготривалі ключі *Користувача 1* зберігаються в базі даних з обліковими записами користувачів. Отримавши запит від клієнта Kerberos, установленого на робочій станції *Користувача 1*, *KDC* звертається до своєї бази даних, знаходить у ній обліковий запис потрібного користувача і витягує з відповідного йому поля довготривалий ключ *Користувача 1*.

Такий процес – обчислення однієї копії ключа за паролем і витягування другої його копії з бази даних – виконується лише один раз за сеанс, коли користувач входить у мережу вперше. Відразу ж після отримання пароля користувача й обчислення довготривалого ключа клієнт *Kerberos* робочої станції запитує сеансовий мандат і сеансовий ключ, що використовуються в усіх подальших транзакціях із *KDC* протягом поточного сеансу роботи в мережі.

На запит користувача *KDC* відповідає спеціальним сеансовим мандатом для самого себе, який називають мандатом на видання

мандатів (*ticket-granting ticket*), або мандатом *TGT*. Як і звичайний сеансовий, мандат *TGT* містить копію сеансового ключа для зв'язку служби (центру *KDC*) із клієнтом. Повідомлення з мандатом *TGT* також містять копію сеансового ключа, за допомогою якої клієнт може зв'язатися з *KDC*. Мандат *TGT* шифрується за допомогою довгострокового ключа служби *KDC*, а клієнтська копія сеансового ключа – за допомогою довгострокового ключа користувача.

Отримавши відповідь від служби *KDC* на свій первинний запит, клієнт дешифрує свою копію сеансового ключа, використовуючи для цього копію довготривалого ключа користувача зі своєї кеш-пам'яті. Після цього довготривалий ключ, отриманий з пароля користувача, можна вилучити з пам'яті, оскільки він більше не знадобиться: подальший зв'язок із *KDC* шифруватиметься за допомогою отриманого сеансового ключа. Як і всі інші сеансові ключі, він має тимчасовий характер і є дійсним до закінчення терміну дії мандата *TGT* або до виходу користувача із системи. З цієї причини такий ключ називають сеансовим ключем реєстрації (*logon session key*).

З погляду клієнта мандат *TGT* майже нічим не відрізняється від звичайного. Перед під'єднанням до будь-якої служби клієнт, передусім звертається до кеш-пам'яті посвідчень і дістає звідти сеансовий мандат потрібної служби. Якщо його немає, то він починає шукати в цій самій кеш-пам'яті мандат *TGT*. Знайшовши його, клієнт отримує звідти ж відповідний сеансовий ключ реєстрації та готує з його допомогою аутентифікатор, який разом з мандатом *TGT* надсилає до центру *KDC*. Одночасно туди відправляється запит на сеансовий мандат для необхідної служби. Інакше кажучи, організація безпечного доступу до *KDC* нічим не відрізняється від організації такого доступу до будь-якої іншої службі домену – вона вимагає сеансового ключа, аутентифікатора і мандата (у цьому разі мандата *TGT*).

З погляду служби *KDC*, мандати *TGT* дозволяють прискорити обробку запитів на отримання мандатів, заощадивши кілька наносекунд на пересиланні кожного з них. Центр розподілу ключів *KDC* звертається до довготривалого ключа користувача тільки один раз, коли надає клієнту початковий мандат на видання мандата. В усіх подальших транзакціях із цим клієнтом центр *KDC* дешифрує мандати *TGT* за допомогою власного довгострокового ключа і витягує з нього сеансовий ключ реєстрації, який використовує для перевірки автентичності аутентифікатором клієнта.

3.9. Висновки

1. Захист відіграє в розподілених системах надзвичайно важливу роль. Розподілені системи мають надавати користувачам і розробникам механізми, що забезпечують реалізацію різноманітних правил захисту. Розробка і правильне застосування механізмів забезпечення захисту в розподілених системах є складною інженерною задачею.

2. Розподілені системи повинні мати засоби для організації захищених каналів зв'язку між процесами. Захищений канал надає засоби взаємної аутентифікації сторін, засоби підтримки конфіденційності, захищає повідомлення під час пересилання від фальсифікації. Це означає, що ніхто, крім сторін, що з’єдналися, не в змозі читати передані каналом повідомлення.

3. Одним із суттєвих питань, яке варто вирішити під час проектування, є вибір серед винятково симетричної криптосистеми, яка грунтується на спільному використанні секретних ключів, та сполученням її із системою з відкритим ключем. На практиці криптографія з відкритим ключем використовується для розсилання загальних секретних сеансових ключів.

4. Друге питання захисту розподілених систем – це контроль доступу, або авторизація. Авторизація потрібна для захисту ресурсів, щоб тільки процеси, які мають відповідні права доступу, могли одержувати доступ до відповідних ресурсів і використовувати їх. Після аутентифікації процесу завжди відбувається контроль доступу.

5. Розрізняють два способи реалізації контролю доступу: 1) кожний ресурс може підтримувати власний список доступу, в якому називаються права доступу всіх користувачів або процесів; 2) процес може мати сертифікат, який точно встановлює його права на визначений набір ресурсів. Основна перевага сертифікатів полягає в тому, що один процес може легко передати свій талон другому, делегувавши свої права доступу. Однак сертифікати мають і недолік – їх зазвичай не просто відкликати.

6. Ще одне питання захисту розподілених систем – це керування захистом, зокрема два важливих аспекти – керування ключами і керування авторизацією. Керування ключами охоплює поширення криптографічних ключів, у якому значну роль відіграють сертифікати, які видає довірена третя сторона. Для керування авторизацією застосовують сертифікати атрибутів і делегування.

7. Важливою сферою використання розподілених систем є електронні платіжні системи, за допомогою яких сторони можуть зв'язуватися через глобальні мережі. Особливу увагу часто приділяють забезпеченню визначеного рівня анонімності покупця, що важливо як у традиційних розрахункових системах на основі готівки, так і в електронних платіжних системах.

8. Протокол *Kerberos* вирізняється гнучкістю та ефективністю використання, а також забезпечує підвищений рівень безпеки. З розвитком Internet, локальних мереж, віртуальних приватних мереж, електронної комерції цей протокол став одним з тих, які відповідають усім вимогам безпеки в сьогоднішньому інформаційному середовищі. З

огляду на це розробники програмного забезпечення приділяють йому підвищену увагу, постійно його вдосконалюючи, а компанія Microsoft, починаючи з операційної системи Windows2000, зробила цей протокол основним протоколом аутентифікації.

3.10.Запитання для самоконтролю

1. На які дві незалежні частини можна поділити системи захисту в розподілених системах?
2. Як визначити надійність розподілених систем?
3. Що таке цілісність інформації?
4. Чим відрізняється аутентифікація від ідентифікації?
5. Дайте визначення поняттю «авторизація».
6. Які вимоги ставлять розподілені системи до проміжного рівня?
7. Що таке фокус керування?
8. Опишіть багаторівневу систему захисту.
9. Роз’ясніть поняття «закрита репликація серверів».
10. Які завдання має виконувати брандмауер?
11. Які два типи брандмауерів вам відомі?
12. Що таке шлюз прикладного рівня?
13. Дати визначення поняттю «контроль доступу».
14. Що таке CRL?
15. Що таке моделі довіри?
16. Які є способи організації платіжних систем?
17. Що передбачає обмін ключами за Диффі-Хеллманом?
18. Чим підписують відкритий ключ та ідентифікатор?
19. Охарактеризуйте електронні платіжні системи у разі використання авізо.

4. РОЗПОДІЛЕНІ СИСТЕМИ ОБ'ЄКТІВ

4.1. Технології побудови розподілених систем

Основними технологіями, які використовуються розробниками інформаційних систем, насьогоді стали технології глобальної мережі Internet і World Wide Web (*WWW*).

Спочатку WWW створювався тільки як засіб, що надає графічний інтерфейс в Internet і спрощує доступ до інформації, розподіленої по мільйонах комп'ютерів у всьому світі. Основними інформаційними компонентами були сторінки, вузли, браузери і сервери Web. Користувачам було надано можливість навігації по Internet з використанням технології гіпертексту, підтримуваної протоколом HTTP (*Hypertext Transfer Protocol*) і стандартом мови HTML (*Hypertext Markup Language*).

З появою CGI (*Common Gateway Interface*) вирішено проблему обміну інформацією між сервером Web і такими програмами, як бази даних, які не можуть безпосередньо обмінюватися даними з браузерами Web. У результаті цього з'явилася можливість реалізувати інтерактивну взаємодію кінцевого користувача з програмами, які містяться на стороні Web-сервера та обробляють інформацію, введену користувачем у браузері та повертають сформовану HTML-сторінку. Багато з наявних рішень щодо доступу до БД у середовищі Internet застосовують даний підхід.

Поява мови Java надала розробникам інформаційних систем абсолютно нові технологічні можливості для побудови прикладних програм у середовищі Internet/Intranet (не варто розглядати Java тільки як частину технології WWW, оскільки вона дозволяє вирішувати більш складні завдання, ніж технологія, що грунтується на мові HTML, протоколі HTTP і CGI).

Є суттєві труднощі, пов'язані з взаємодією інформаційних систем, в основу яких покладено технологію WWW, досі проблематичною є інтеграція цих систем між собою і з наявними великими інформаційними системами.

Наявні різні способи вирішення цієї проблеми, в основі яких лежить модель розподілених об'єктних технологій, вибір якої багато в чому визначає характеристики інформаційної системи, яку створюють.

Розподілена інформаційна система є сукупністю програмних компонент, які взаємодіють. Кожен такий компонент має програмний модуль, що виконується в межах окремого процесу. Використання об'єктно-орієнтованого підходу під час створення крупних інформаційних систем дозволяє розглядати компоненти інформаційної системи на різних рівнях абстрагування як об'єкти, кожен з яких має певну лінію поведінки. Взаємодія таких об'єктів здійснюється на основі певного середовища взаємодії, головною метою створення якого є реалізація механізму обміну повідомленнями в контексті гетерогенних розподілених середовищ, існування яких в певних межах характерно для більшості великих організацій.

Побудова середовища взаємодії - один із складних етапів розробки інформаційної системи. Створення розробниками інформаційних систем власного, унікального середовища взаємодії об'єктів призводить, з одного боку, до стрімкого збільшення витрат на реалізацію проекту побудови інформаційної системи, а з іншого, до функціональної неповноти та економічної збитковості отриманого рішення.

Дослідження проектів створення інформаційних систем дозволяють дійти висновку, що для того, щоб уникнути невиправданих витрат на розробку власного, унікального інформаційного середовища, необхідно використовувати вже наявні програмні продукти, які

належать до рівня проміжного програмного забезпечення (*middleware*) і реалізують необхідні функції середовища взаємодії.

Проте не всі програмні середовища рівня *middleware* можуть використовуватися як середовище взаємодії об'єктів великої інформаційної системи. Це обумовлено тим, що однією з основних вимог до такої системи є використання програмних продуктів і технологій, які відповідають міжнародним і промисловим стандартам відкритих інформаційних систем, якими є стандарти DCOM/COM, CORBA, Web-сервісу.

Розподілені об’єкти. В основу побудови технології розподілених систем об’єктів покладено можливість розподілу об’єктів глобальною мережею. Розподілені об’єкти всі аспекти розподілу мережею приховують за своїм інтерфейсом, довільно розташовуючись в Інтернет-мережі. Такий підхід має суттєве значення для побудови розподілених систем, зокрема розповсюджених розподілених середовищ.

Одним із найвідоміших є промисловий стандарт розподілених систем CORBA, який використовують оператори зв'язку під час створення програмних платформ надання послуг.

Другим прикладом є технологія DCOM корпорації Microsoft. DCOM можна розглядати як програмне середовище проміжного рівня, реалізоване поверх різних операційних систем Windows, починаючи з Windows 95. Фактично все прикладне програмне забезпечення Windows застосовує функціональність, яка надається DCOM. DCOM є поширеною технологією для побудови розподіленого середовища проміжного рівня.

Наступним кроком в розвитку програмних технологій проміжного рівня стала технологія Web-сервісів, в якій поєднано переваги двох попередніх технологій, її покладено в основу *сервіс-орієнтованої архітектури* (*Service Oriented Architectures, SOA*).

Крім відомих комерційних технологій проміжного рівня є низка дослідницьких підходів щодо побудови розподілених систем об'єктів, серед яких вирізняють систему Globe, яка застосовує оригінальну технологію проміжного рівня, що може окремо зберігати та реплікувати стан розподілених об'єктів на декількох машинах.

Розглянемо порівняльну характеристику розподілених середовищ, починаючи з найбільш важливих концепцій побудови системи, а саме її об'єктної моделі й основних служб, які вона підтримує. Далі обговоримо сім основних принципів реалізації системи: зв'язок, процеси, іменування, синхронізацію, несуперечливість і реплікацію, відмовостійкість, захист.

Система, яка побудована за технологією розподілених об’єктів, утворюється з набору компонент (об’єктів), які взаємодіють між собою. Об’єкти розкидані мережею і виконуються окремо один від одного.

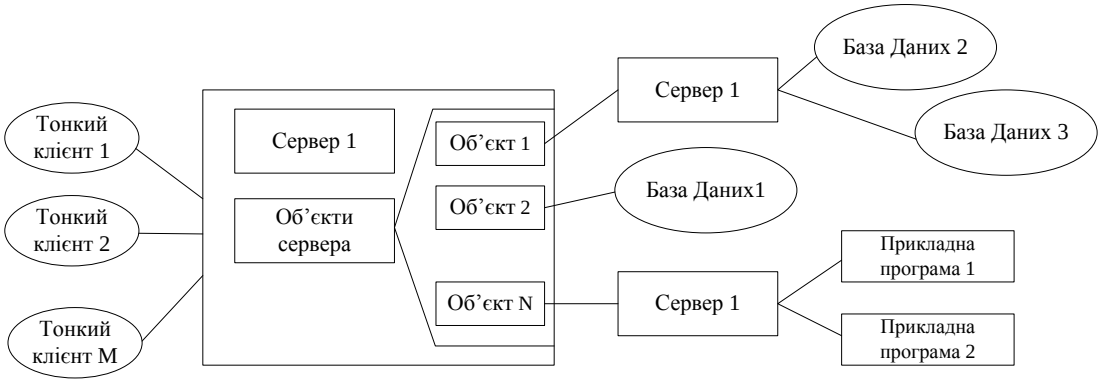


Рис. 4.1. Структура розподіленої системи

У розподілених системах об'єктів поняття «об'єкт» відіграє ключову роль для реалізації прозорості розподілу. Вважати об'єктами можна різні ресурси мережі, наприклад, клієнти отримують служби і ресурси у формі об'єктів, до яких вони можуть звертатися.

Переваги використання розподілених систем об'єктів:

- скорочення часу розробки (ізольована розробка);
- скорочення кількості помилок;
- повторне використання програмних компонент;
- полегшується майбутній реінженірінг системи;

- можливість побудови «тонких» клієнтів.

Повторне використання коду. Розподілені об’єкти дозволяють швидко та ефективно створювати багатофункціональні прикладні програми, використовуючи вже існуючі plug-and-play компоненти, що помітно знижує вартість побудови нової системи.

Ізольована розробка. Завдяки модульній основі розподілене прикладне програмне забезпечення дозволяє здійснити розробку або заміну модулів (компонент), ізольованих один від одного.

Вся система поділяється на замкнені, автономні модулі, робота щодо створення яких може виконуватись окремо, такі модулі можуть також взаємодіяти з іншими модулями системи. Для цього вони мають підтримувати протоколи та інтерфейси, які визначають принципи їх взаємодії. Оскільки методи, існуючі в модулях, ізольовані від методів інших модулів, то їх розробляють незалежно, тобто стан реалізації компонент не залежить від стану реалізації коду в інших частинах системи. Взаємодія між різними модулями відбувається через встановлені протоколи та інтерфейси.

Супровід прикладного програмного забезпечення. Завдяки модульному підходу заміна певної функціональної частини не вимагає глобальної перебудови всієї системи. Навпаки, заміна коду відбувається тільки в модулях, які цього вимагають, що є як простішим, так і значно швидшим. Знижується також і ризик виникнення помилок під час заміни коду, тому що зміни здебільшого стосуються внутрішньої частини об'єктів.

Тонкі клієнти. У розподіленій системі є можливість перенести всю функціональну логіку на її серверну частину. В цьому разі прикладні програми-клієнти, з якими взаємодіє користувач, розробляють невеликими та потребуючими незначних обчислювальних ресурсів. Системні ресурси користувача не перевантажують, а всю функціональну логіку реалізують на високопотужному сервері (або на

групі серверів). У такому разі клієнт має доступ до практично необмеженої кількості сховищ даних та інших об'єктів, з'являється можливість створення нересурсомістких компонент, придатних для швидкого завантаження мережею Internet, а також запуску на комп'ютері клієнта (до таких прикладних програм відносять аплети або *ACTIVE-X* компоненти).

Першими технологіями, які підтримували системи розподілених об’єктів були найпростіші технології типу *RPC* (*Remote Procedure Call*). Подальшим розвитком *RPC* у мові Java стала технологія *RMI* (*Remote Method Invocation*).

4.2. Common Object Request Broker Architecture

Технологію розподілених систем об'єктів розглянемо з погляду узагальненої архітектури брокера об'єктних запитів (*Common Object Request Broker Architecture, CORBA*), яка є не стільки технологією побудови розподілених систем, скільки її специфікацією. Подібні специфікації розроблені некомерційною спеціалізованою організацією, групою керування об'єктами (*Object Management Group, OMG*). Основною метою OMG під час розробки CORBA було створення технології побудови розподіленої системи, яка б не мала більшості недоліків міжопераційної сумісності у разі інтеграції мережного прикладного програмного забезпечення. Перші специфікації CORBA з’явилися на початку 90-х років.

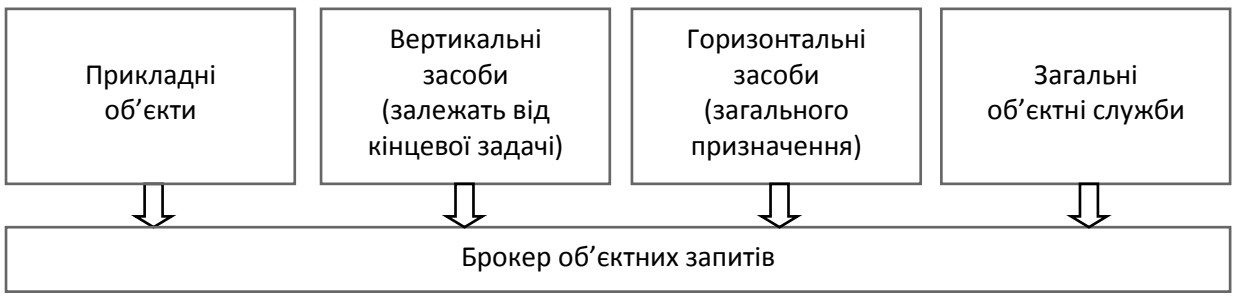


Рис. 4.3. Глобальна архітектура CORBA

Глобальну архітектуру CORBA у відповідності до моделі OMG представлено на рис. 4.3, вона містить чотири групи архітектурних елементів, пов'язаних з *брокером об'єктних запитів* (*Object Request Broker, ORB*), який утворює ядро будь-якої розподіленої системи, побудованої за допомогою технології CORBA. ORB відповідає за підтримку зв'язку між об'єктами та їх клієнтами, приховуючи особливості, пов'язані з розподілом і різнорідністю системи. В багатьох розподілених системах ORB реалізується у вигляді множини бібліотек, які компонуються з прикладним програмним забезпеченням клієнта і сервера, надаючи їм базові служби зв'язку.

Окрім об'єктів, які є частинами конкретних прикладних програм, в модель також входять інструментальні засоби CORBA (*CORBA facilities*), які поєднують внутрішні служби CORBA і поділяються на дві різні групи: горизонтальні та вертикальні засоби.

Горизонтальні засоби (*horizontal facilities*) є високорівневими службами загального призначення, які не залежать від прикладної області програм, які їх використовують. Подібні служби обслуговують інтерфейс, який призначений для користувача, процеси керування інформацією, керування системою і керування завданнями, що потрібно для виконання робочих потоків.

Вертикальні засоби (*vertical facilities*) – це високорівневі служби, призначені для конкретних предметних сфер застосування інформаційної системи, таких як електронна комерція, банківська справа, виробництво та ін.

4.2.1. Об’єктна модель

Технологія CORBA використовує модель віддалених об’єктів, за якої реалізація об’єктів відбувається в адресному просторі сервера.

Специфікації CORBA не визначають як обов’язкову вимогу, щоб об'єкти реалізовувалися виключно як віддалені, проте фактично всі системи на базі CORBA підтримують тільки цю модель. Крім того, специфікації часто рекомендують, щоб розподілені об'єкти в CORBA були реалізовані в формі віддалених об’єктів.

Об'єкти і служби описують за допомогою мови визначення інтерфейсів (*Interface Definition Language, IDL*) CORBA. Мова IDL у CORBA подібна іншим мовам визначення інтерфейсів, має традиційний для цих мов синтаксис опису методів та їх параметрів. Описати семантику IDL для CORBA неможливо, тому що інтерфейс – це набір методів, а об'єкт сам визначає, які інтерфейси він реалізує.

Специфікації інтерфейсу можна задати тільки за допомогою IDL. В таких системах, як *Distributed COM* і *Globe*, інтерфейси визначають на більш низьких рівнях у вигляді таблиць. Такі бінарні інтерфейси (*binary interfaces*) не залежать від мов програмування. В CORBA необхідно повністю задати правила відображення специфікації на мові IDL на використовувану мову програмування. Такі правила наявні для низки мов, включаючи C, C++, Java, Smalltalk, Ada і COBOL.

Враховуючи, що система CORBA організована як набір клієнтів і серверів об'єктів, її загальну організацію наведено на рис. 4.4.

Основою кожного процесу взаємодії в CORBA на стороні клієнта або на стороні сервера є посередник запитів до об’єктів (*Object Request Broker, ORB*). ORB можна розглядати як систему часу виконання, яка відповідає за базові функції зв'язку між клієнтом і об'єктом. Ці базові функції зв'язку гарантують звернення до сервера об'єктів, а потім повернення клієнтові відповідей сервера.

З погляду процесу, брокер ORB надає лише деякі послуги, одна з яких – обробка посилань на об'єкти. Вигляд цих посилань зазвичай залежить від конкретного брокера ORB, тому будь-який брокер ORB

надає операції для маршалінга і демаршалінга посилань на об'єкти під час обміну між процесами, а також операції для порівняння посилань.

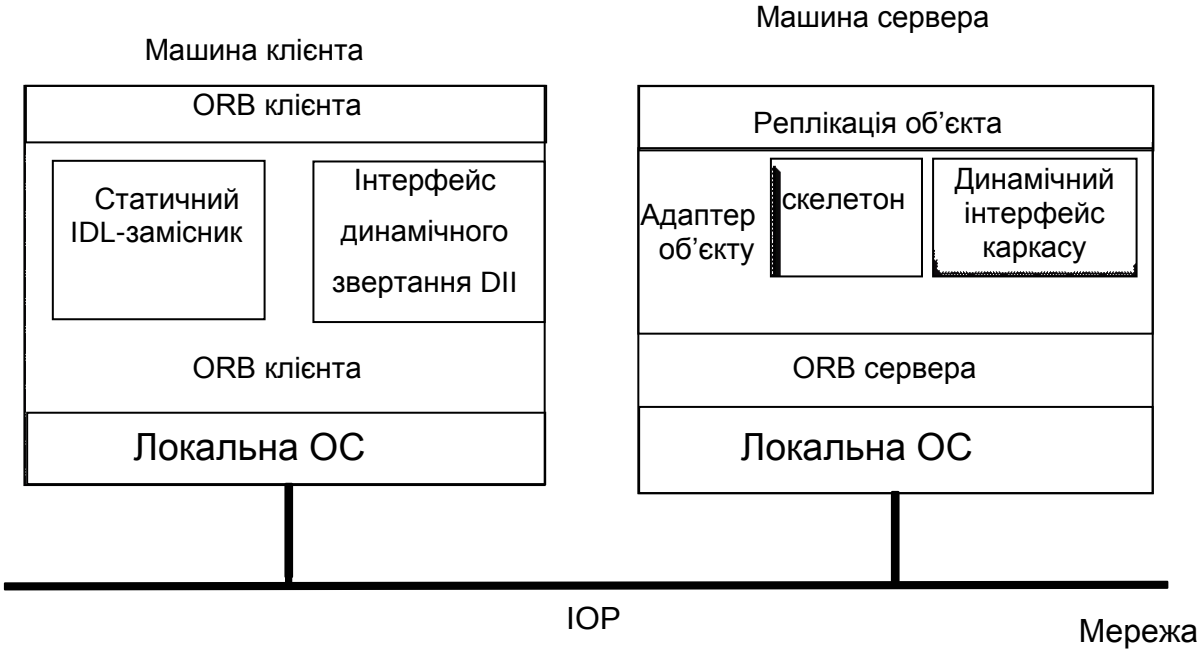


Рис. 4.4. Загальна організація системи CORBA

Важливою з погляду функціональності операцією брокера ORB є початковий пошук доступних для процесу служб, за якої брокер надає дані для отримання початкового посилання на об'єкт, який реалізовує конкретну службу CORBA. Так, наприклад, для використання служби іменування необхідно, щоб процес знав, як на неї посилатися. Подібну задачу ініціалізації також необхідно вирішувати і для інших служб.

Крім інтерфейсу клієнти і сервери в ORB бачать лише заглушки, які оброблюють звертання до відповідних об'єктів. Клієнтське програмне забезпечення містить замісника, призначеного для реалізації того ж інтерфейсу, який використовує об'єкт, тому замісник – це клієнтська заглушка, єдине призначення якої – виконати маршалінг (тобто упаковку) звертання в запит і переслати цей запит серверу. Після отримання відповіді з сервера виконується її демаршалінг (розпаковування) і передача клієнту.

Відзначимо, що інтерфейс між замісником і ORB не може бути стандартизованим. CORBA передбачає, що всі інтерфейси створено

мовою IDL, тому компілятор IDL генерує код, необхідний для забезпечення зв'язку між клієнтським і серверним брокерами ORB.

Проте бувають випадки, коли статично визначені інтерфейси недоступні клієнту. В цьому разі під час виконання необхідно з'ясувати, як виглядає інтерфейс для конкретного об'єкту, щоб мати можливість послідовно реалізовувати запит із звертанням до цього об'єкта. Для цієї мети CORBA надає клієнтам *інтерфейс динамічного виклику* (*Dynamic Invocation Interface, D1I*), який дозволяє створювати запити у ході виконання прикладного програмного забезпечення. DII пропонує базову операцію виклику (*Invoce*), яка отримує параметри посилання на об'єкт (ідентифікатор методу і список вхідних значень), а повертає в результаті значення згідно списку вихідних змінних, який надається стороною, що викликає.

Система CORBA містить адаптер об'єктів (рис. 4.4), який відповідає за передачу вхідних запитів потрібному об'єкту. Реальний демаршалінг (розпаковування даних запиту) на стороні сервера виконується заглушками, які в CORBA називають скелетонами, проте можливою є ситуація, коли реалізація об'єкту бере демаршалінг на себе. Аналогічно заглушкам клієнта, серверні заглушки також можуть компілюватися статично з опису на мові IDL або мати вигляд базового динамічного скелетона. У разі використання динамічного скелетона об'єкт має містити правильну реалізацію звертання до доступних клієнту функцій.

4.2.2. Сховища інтерфейсів і реалізацій

Для того, щоб забезпечити динамічне створення запитів із викликами методів об’єктів, необхідно, щоб у ході виконання процес міг визначити, як виглядає інтерфейс. CORBA надає сховище інтерфейсів (*interface repository*), в якому зберігаються всі визначення

інтерфейсів. В багатьох системах сховище інтерфейсів реалізується у вигляді окремого процесу, який має стандартний інтерфейс зберігання та видачі визначень інтерфейсів. Сховище інтерфейсів також може розглядатися як частина CORBA, яка сприяє роботі механізмів перевірки типів під час виконання.

Коли компілюється визначення інтерфейсу, компілятор IDL призначає ідентифікатор зберігання (*repository identifier*) цього інтерфейсу. Цей ідентифікатор – основний засіб отримання визначення інтерфейсу зі сховища. Ідентифікатор за замовчанням формується з імені інтерфейсу і його методів, що не гарантує його унікальності. Якщо висувається вимога щодо наявності унікальності, стандартний ідентифікатор можна змінити.

Враховуючи, що всі визначення інтерфейсів зберігаються у сховищі інтерфейсів згідно з їх синтаксисом на мові IDL, стає можливим організувати кожне визначення стандартним чином, тому сховища інтерфейсів у системах CORBA мають однакові операції навігації. (За термінологією баз даних це означає, що концептуальна схема, асоційована зі сховищем інтерфейсів, однакова для всіх сховищ).

Окрім сховища інтерфейсів система CORBA пропонує також сховище реалізацій (*implementation repository*). Концептуальне сховище реалізацій містить все необхідне для реалізації й активації об'єктів. Оскільки подібна функціональність спочатку пов'язана з самим брокером ORB і базовою операційною системою, то створити стандартну реалізацію проблематично.

Сховище реалізацій, крім того, тісно пов'язано з організацією і реалізацією серверів об'єктів. Задача адаптера об’єктів – активізувати об’єкти за допомогою їх запуску в адресному просторі сервера таким чином, щоб до їх методів можна було звертатися. Отримавши

посилання на об'єкт, адаптер об'єктів зв'язується зі сховищем реалізацій, щоб знайти ту реалізацію, яка йому потрібна.

Приклад. Розглянемо два приклади, які показують, наскільки тісно пов'язано сховище з брокером ORB і платформою, на якій він працює.

1) Сховище реалізацій може підтримувати таблицю, яка вказує на можливість запуску нового сервера, номер порту, який може бути призначений новому серверу для прослуховування конкретним об'єктом. Сховище, крім того, міститиме інформацію про виконуваний файл (тобто програму), який сервер має завантажити в пам'ять і виконати.

2) Необхідності запуску окремого сервера може і не виникнути, досить буде скомпонувати наявний сервер з деякою бібліотекою, що містить указаний метод або об'єкт. Інформація про це може міститися в сховищі реалізацій.

4.2.3. Служби CORBA

Важливою складовою структурної моделі CORBA є набір служб CORBA, які є службами загального призначення, не залежними від прикладного програмного забезпечення. Типи служб CORBA подібні типам служб операційних систем. Повний список служб CORBA наведено в табл. 4.1. Нажаль, не завжди можна провести межу між різними службами, оскільки їх функціональність часто перетинається. Коротко розглянемо кожну із служб, щоб у подальшомупорівняти їх із службами DCOM і Globe.

Таблиця 4.1. Служби CORBA

Служба	**Опис**
Служба колекцій	Засоби групування об'єктів у списки, черги, множини й т. інш.
Служба запитів	Засоби для декларування запитів до наборів об’єктів
Служба паралельного доступу	Засоби забезпечення паралельного доступу до об'єктів, які спільно використовуються
Служба транзакцій	Прості й вкладені транзакції для виклику методів різних об'єктів
Служба подій	Засоби асинхронної взаємодії з застосуванням механізму подій

Служба повідомлень	Розширені засоби асинхронної взаємодії з застосуванням механізму подій
Служба зовнішніх зв'язків	Засоби маршалиінгу і демаршалінгу об’єктів
Служба життєвого цикла	Засоби створення, видалення, копіювання і переміщення об'єктів
Служба ліцензувань	Засоби для приєднання до об'єкту ліцензії
Служба іменування	Засоби іменування об'єктів в межах системи
Служба властивостей	Засоби приєднання до об'єкту пар (атрибут, значення)
Служба обміну	Засоби публікації та пошуку служб, потрібних об'єкту
Служба збереження	Засоби тривалого зберігання об'єктів
Служба відношень	Засоби визначення взаємодії між об'єктами
Служба захисту	Механізми створення захищених каналів, авторизації й аудиту поточного часу із заданою помилкою

Служба колекцій (*collection service*) в залежності від природи групи використовує різні механізми доступу. Так, наприклад, списки можна переглядати поелементно за допомогою механізму, який називають ітератором. Є також і засоби пошуку об'єктів за заданим ключовим значенням. Служба колекцій аналогічна бібліотеці класів об'єктно-орієнтованих мов програмування.

Служба запитів (*query service*) надає дані для створення колекцій об'єктів, до яких можна звертатися, використовуючи декларативні мови запитів. Запит може повертати посилання на об'єкт або колекцію об'єктів. Служба запитів розширює службу колекцій можливістю створення розширених запитів та відрізняється від служби колекцій тим, що передбачає створення колекцій різних типів.

Служба паралельного доступу (*concurrency service*) надає розширений механізм блокувань, який клієнт може застосовувати під час доступу до загальних об'єктів. Ця служба може використовуватися для реалізації транзакцій, які виділяють в окрему службу.

Служба транзакцій (*transaction service*) дозволяє клієнту визначати ряд звертань до методів декількох об'єктів у вигляді єдиної транзакції, підтримує прості та вкладені транзакції.

Зазвичай клієнти звертаються до методів, після чого очікують результату. Для підтримки асинхронної взаємодії CORBA містить *службу подій* (*event service*), за допомогою якої клієнт і об’єкт можуть переривати роботу у разі, коли відбувається певна подія. Додаткові засоби асинхронної взаємодії використовує окрема *служба повідомлень* (*notification service*).

Служба зовнішніх зв’язків (*extemalization service*) здійснює маршалінг об’єктів таким чином, щоб їх можна було зберегти на диск або передати мережею, та є подібною до засобів серіалізації Java, які дозволяють записувати об’єкти в потік даних у вигляді послідовності байтів.

Служба життєвого циклу (*life cycle service*) дозволяє створювати, знищувати, копіювати та переміщувати об’єкти. Ключова концепція такої служби – *об’єкт-фабрика* (*factory object*), який є спеціальним об'єктом, що використовується для створення інших об’єктів. Практика показує, що окремі служби вимагають тільки створення об’єктів, а знищення, копіювання і переміщення об'єктів часто зручніше визначати всередині самих об'єктів, тому що на ці операції впливає стан об'єкту, тобто спосіб здійснення цих операцій залежить від об'єкта.

Служба ліцензування (*licensing service*) дозволяє розробникам приєднувати до своїх об'єктів ліцензії й підтримувати певні правила ліцензування. Ліцензії визначають права клієнта на використання об'єкта. Так, наприклад, ліцензія, приєднана до об'єкта, може вказувати, що об'єкт дозволяється використовувати одночасно лише одному клієнту. Друга ліцензія може забезпечувати автоматичну деактивацію об'єкту після закінчення певного часу.

Для об'єктів, що мають змістовні імена, CORBA підтримує окрему *службу іменування* (*naming service*), яка відображає ці імена на ідентифікатори об’єктів. Основні засоби опису об’єктів зібрані в

окремій *службі властивостей* (*property service*), яка дозволяє клієнтам асоціювати з об'єктами пари (атрибут, значення), які не є частинами стану об'єкту, а використовуються саме для його опису, тобто надають інформацію про об'єкт, яка не є його частиною. З останніми двома службами пов’язана *служба обміну* (*trading service*), яка дозволяє об’єктам інформувати про те, що вони можуть виконувати (за допомогою їх інтерфейсів), а клієнтам здійснювати пошук потрібних служб, використовуючи спеціальну мову, яка підтримує опис обмежень.

Окрема *служба зберігання* (*persistence service*) містить засоби збереження інформації на диску у вигляді об'єктів, передбачаючи прозорість зберігання – клієнту не потрібно явно передавати дані між диском і оперативною пам'яттю.

Перераховані служби не надають засобів явного визначення взаємодії між процесами. Такі засоби застосовує *служба відношень* (*relationship service*), яка підтримує організацію об’єктів згідно з концептуальною схемою, аналогічно схемам відношень між об'єктами в базах даних.

Захист забезпечується *службою захисту* (*security service*), яка подібна до систем захисту SESAME і Kerberos. Служба захисту в CORBA підтримує засоби аутентифікації, авторизації, аудиту, захищеного зв'язку й адміністрування.

CORBA підтримує *службу часу* (*time service*), яка повертає поточний час з деякою заданою помилкою.

Служби CORBA створено з використанням об'єктної моделі CORBA як базової, тому всі служби описані мовою IDL, але між описом і реалізацією інтерфейсів існує відмінність. Важливий принцип побудови CORBA полягає в тому, що служби мають бути прості й мінімальні за об'ємом пам'яті, необхідної для їх застосування.

4.3. Distributed Component Object Model

Розглянемо розподілену систему об'єктів, яка застосовує розподілену модель компонентних об’єктів (*Component Object Model, COM)* корпорації Microsoft. Модель DCOM (*Distributed COM*) створена з використанням моделі COM, якою є технологія, покладена в основу різних версій операційних систем Windows від Microsoft, починаючи з Windows 95. В моделі DCOM відсутня спроектована архітектура з мінімальним набором базових елементів, з яких утворюють компоненти та служби.

Метою створення COM була підтримка розробки компонентів, які могли б динамічно активізуватися і взаємодіяти один з одним. **Компонентом в COM** є код, який виконується та міститься у динамічно компонованій бібліотеці (*Dynamic-link library, DLL*) або в програмі, яка виконується.

Модель COM використовують у вигляді бібліотек, які компонуються з процесом. Спочатку модель було розроблено для підтримки **складних документів** (*compound documents*), які побудовані з різнорідних частин, таких як текст (форматований), зображення, електронні таблиці та т.інш. Кожна з цих частин може бути відредагована за допомогою асоційованого з нею прикладного програмного забезпечення.

Для підтримки значної кількості складних документів використовують узагальнений метод для розділення їх окремих частин або об'єднання в єдиний об’єкт. Спочатку це виконувалось за **технологією зв'язування і вкладання об'єктів** (*Object Linking and Embedding, OLE*). Першу версію OLE застосовували для передачі інформації між частинами документа, але це був негнучкий спосіб обміну повідомленнями. Незабаром її було змінено наступною версією,

яка побудована на базі більш гнучкого механізму COM зі структурою, показаною на рис. 4.5.

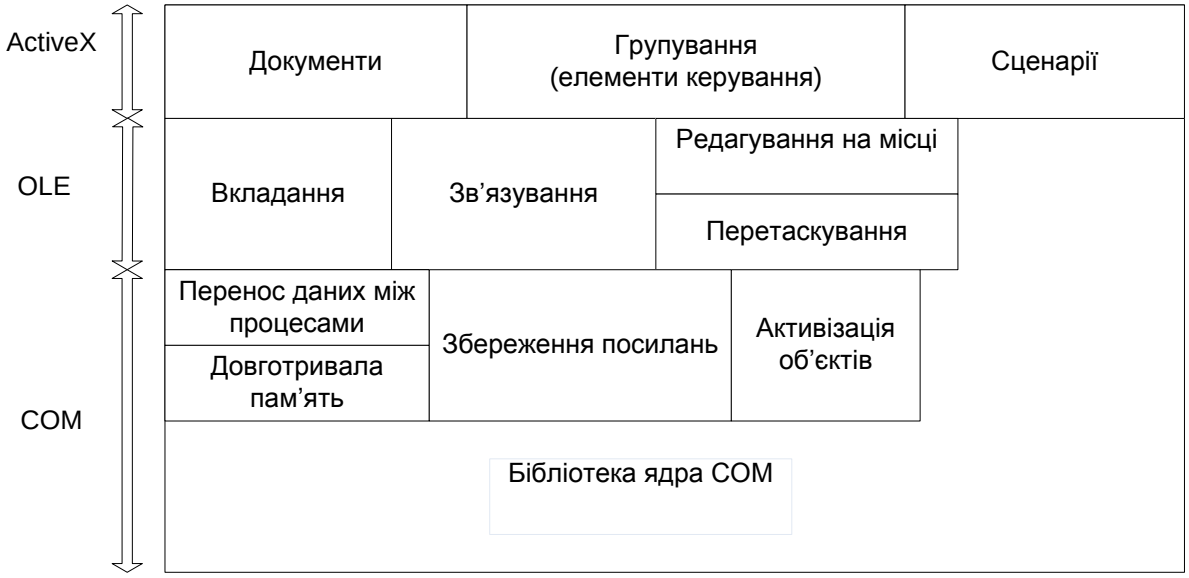


Рис. 4.5. Загальна структура ActiveX, OLE і COM

Технологія ActiveX (рис. 4.5) стала подальшим розвитком технології OLE за рахунок введення таких новації, як гнучка здатність компонентів виконуватися в різних процесах, підтримка сценаріїв і стандартне групування об'єктів в елементи керування ActiveX.

В моделі DCOM процес став здатним працювати з компонентами, розміщеними на другій машині, проте базовий механізм обміну повідомленнями між компонентами, прийнятий в DCOM, дуже часто співпадає з відповідними механізмами COM, тобто відмінність між ними приховано за рахунок використання різних інтерфейсів. DCOM завдяки цьому реалізує прозорість доступу.

4.3.1. Об'єктна модель

DCOM використовує модель віддалених об'єктів, тобто об'єкти в DCOM можуть розміщуватися як в одному процесі з клієнтом, так і на одному коп'ютері з ним, а також в процесі, який виконується на віддаленій машині.

Як і в CORBA, об'єктна модель в DCOM побудована за рахунок реалізації інтерфейсів, тому **об'єкт DCOM** є певною реалізацією одного або декількох інтерфейсів. На відміну від CORBA в DCOM застосовано тільки *бінарні інтерфейси* (*binary interfaces*), які є таблицею з посиланнями на реалізації методів, що є частиною інтерфейсу. Для визначення інтерфейсу використовують спеціальну *мову визначення інтерфейсу* (*Interface Defenition Language, IDL*).

У DCOM така мова (*Microsoft Interface Defenition Language, MIDL*) генерує бінарні інтерфейси стандартного формату, перевагою яких є те, що вони не залежать від мови програмування. За технологією CORBA кожного разу для підтримки певної мови програмування необхідно відображати опис IDL на конструкції цієї мови, що непотрібно у разі використання бінарних інтерфейсів. Відмінність між цими двома підходами показано на рис. 4.6.

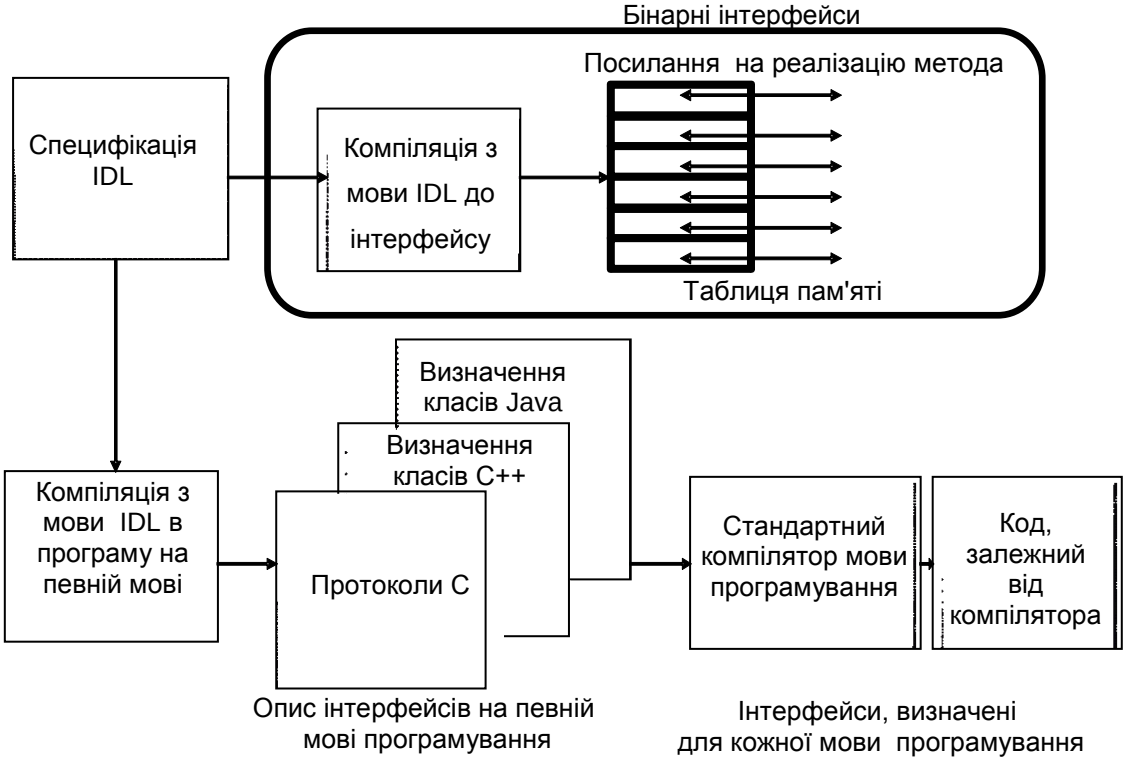


Рис. 4.6. Відмінність між бінарними інтерфейсами та інтерфейсами, визначеними для кожної конкретної мови

Кожен інтерфейс в DCOM має унікальний 128-бітовий ідентифікатор, який називають *ідентифікатором інтерфейсу* (*Interface*

Identifier, IID), що є абсолютно унікальним, тому не існує двох інтерфейсів з однаковим ідентифікатором IID. Ідентифікатор створюють, комбінуючи велике випадкове число, локальний час і адресу мережевого інтерфейсу поточного хосту. Ймовірність того, що два згенерувані індикатори однакові, практично дорівнює нулю.

Об'єкт DCOM створюють як екземпляр класу, для чого необхідно мати доступ до відповідного класу. Для цієї мети DCOM містить *об'єкти класу* (*class objects*), якими можуть бути різноманітні програмні компоненти, які підтримують інтерфейс *IClassFactory*. Цей інтерфейс містить метод *CreateInstance*, подібний до оператора *new* в мовах C++ та Java. Виклик метода *CreateInstance* для певного об'єкта класу створює об'єкт DCOM, який міститиме реалізацію його інтерфейсів, тому об'єкт класу є колекцією об'єктів одного типу, що реалізують один і той же набір інтерфейсів.

Об'єкти, які належать до одного класу, зазвичай вирізняють лише за частиною поточного стану. Завдяки створенню екземплярів об'єкту класу стає можливим звернення до методів інтерфейсів об'єктів. У DCOM до будь-якого об'єкту класу можна звертатися за його глобальним унікальним ідентифікатором класу (*Class Identifier, CLSID*).

Всі об'єкти реалізують стандартний об'єктний інтерфейс *Iunknown*. У разі, коли викликом *CreateInstance* створюють новий об'єкт, то об'єкт класу повертає посилання на цей стандартний інтерфейс. Метод *IUnknown* містить також метод *QueryInterface*, який засобами *IID* повертає посилання на інший інтерфейс, реалізований в об'єкті.

Важлива відмінність DCOM від об'єктної моделі CORBA полягає в тому, що всі об'єкти в DCOM нерезидентні, тому якщо в об'єкта не залишається клієнтів, які посилалися б на нього, то цей об'єкт видаляють. Підрахунок посилань виконують викликаючи методи *Addref* і *Release*, які містить інтерфейс *Iunknown*. Наявність виключно нерезидентних об'єктів унеможливлює зберігання кожним з об'єктів

свого унікального глобально ідентифікатора. З цієї причини об'єкти DCOM можна викликати лише застосовуючи посилання на інтерфейси. Відповідно, для передачі посилання на об'єкт другому процесу необхідно виконати спеціальні перетворення.

DCOM дозволяє застосовувати динамічне звертання до об'єктів. Об'єкти, запити до яких створюватимуться динамічно, тобто під час виконання, мають використовувати реалізацію інтерфейсу *Idispatch*, який подібний до інтерфейсу динамічного виклику *DII* в системі CORBA.

4.3.2. Бібліотека типів і системний реєстр

Еквівалентом сховища інтерфейсів CORBA в DCOM є *бібліотека типів* (*type library*), яку асоціюють з прикладною програмою або другим складним компонентом, що містить різні об’єкти класів. Бібліотека типів може зберігатися в окремому файлі або бути частиною прикладної програми, у будь-якому разі вона потрібна в першу чергу для опису сигнатури методів, які викликають динамічно. Крім того, бібліотеки типів використовують у засобах програмування для відображення на екрані структури інтерфейсів, які розробляють.

Для того, щоб активізувати об'єкт, тобто гарантувати, що його створено та розміщено у процесі, з якого буде виконуватися звертання до методів, у DCOM використовують реєстр Windows у комбінації зі спеціальним процесом – менеджером керування службами (*Service Control Manager, SCM*), який додатково застосовують для запису відображення ідентифікаторів класів (*Class Identifier, CLSID*) на локальні імена файлів, що містять їх реалізації. Щоб створити екземпляр об'єкту, процес спочатку має перевірити, що відповідний об'єкт класу завантажений.

Якщо об'єкт функціонує на віддаленому хості, то у цьому разі клієнт спочатку контактує з менеджером керування службами цього хоста, який є процесом, відповідальним за активізацію об'єктів, це виконується аналогічно з алгоритмом функціонування сховища реалізацій в CORBA. *SCM* віддаленого хоста переглядає свій локальний реєстр, щоб знайти файл, асоційований з *CLSID*, після чого запускає процес, який містить потрібний об'єкт. Сервер виконує маршалінг посилання на інтерфейс і повертає його клієнту, який здійснює демаршалінг посилання і передає його заміснику.

Загальну архітектуру DCOM, яка містить об'єкти класів, реальні об'єкти та замісники, подано на рис. 4.7. Процес клієнта отримує доступ до *SCM* і реєстру, який застосовує для знаходження віддаленого об'єкта, та виконує прив'язку до нього. Клієнт взаємодіє також із замісником, який реалізує інтерфейси цього об'єкту.

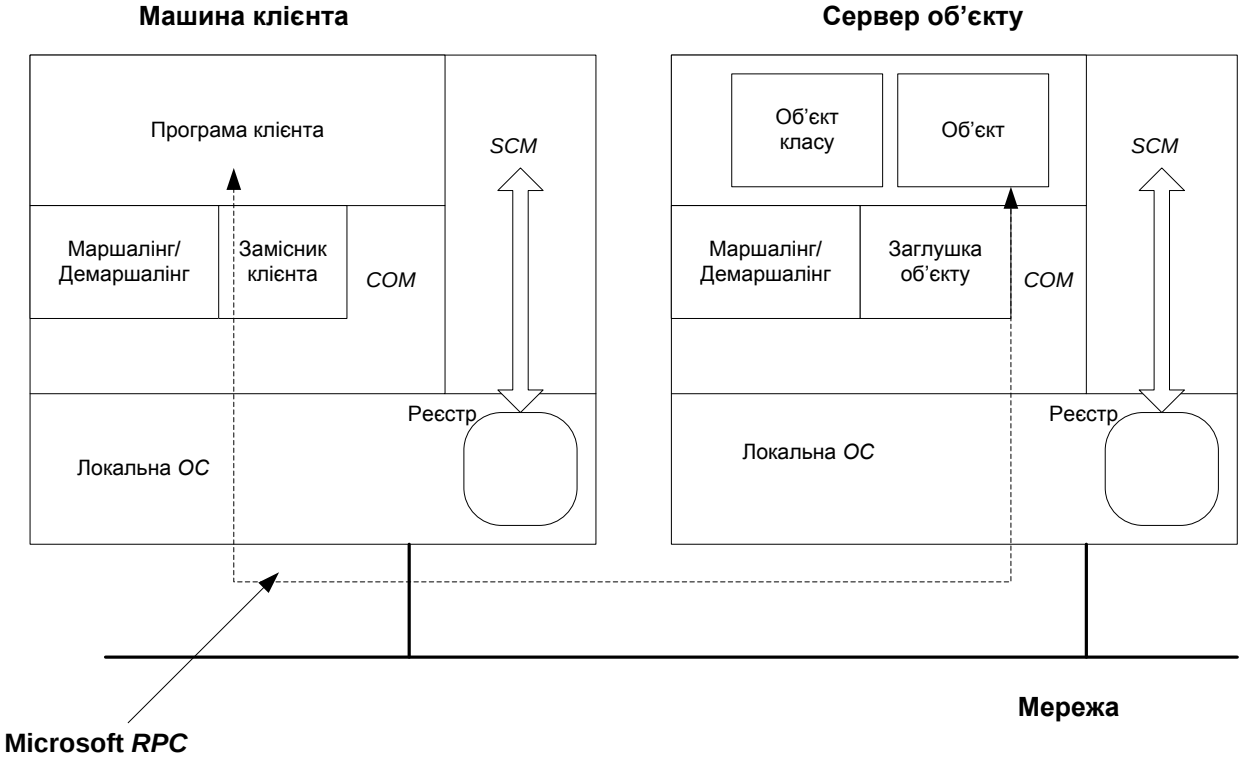


Рис. 4.7. Загальна архітектура DCOM

Сервер об'єктів містить заглушку для маршалінга і демаршалінга звертань, які передаватиме реальному об'єкту. Зв'язок між клієнтом і сервером реалізують, застосовуючи віддалені виклики *RPC*, але можливо використовувати й інші способи зв'язку.

4.3.3. Служби DCOM

Порівняно з моделлю COM модель DCOM розширено додаванням засобів обміну повідомленнями з віддаленими об'єктами. Натомість модель COM було розширено включенням різних зовнішніх служб, які раніше лише доповнювали COM, та названо моделлю COM+. Так, зокрема, COM+ містить розширення, які дозволяють серверу підтримувати одночасно декілька об'єктів, додаткові служби, розташовані на зовнішніх серверах, наприклад, системи черг повідомлень.

Визначити відмінність між COM, DCOM, COM+ та зовнішніми службами іноді на практиці важко, але DCOM є розширенням, яке включає як COM, так і COM+, а також ACTIVE-X. Розглянемо служби, які безпосередньо входять в DCOM (табл. 4.2), у порівнянні зі службами CORBA і Windows 2000, які зазвичай доступні клієнтам моделі DCOM, але не є її частинами.

Таблиця 4.2. Огляд служб CORBA, DCOM і Windows 2000

Служба CORBA	**Служба DCOM/COM+**	**Служба Windows 2000**
Служба колекцій	Об’єкти даних ActiveX	—
Служба запитів	Відсутня	—
Служба паралельного доступу	Паралельні потоки виконання	—
Служба транзакцій	Автоматичні транзакції COM+	Координатор розподілених транзакцій
Служба подій	Події COM+	—
Служба повідомлень	Події COM+	—
Служба зовнішніх зв’язків	Утиліти маршалінга	—
Служба життєвого циклу	Фабрики класів, JIT-активізація	—

Служба ліцензування	Спеціальні фабрики класів	—
Служба іменування	Монікери	Active Directory
Служба властивостей	—	Active Directory
Служба обміну	—	Active Directory
Служба збереження	Спеціальне сховище	Доступ до баз даних
Служба відношень	—	Доступ до баз даних
Служба захисту	Авторизація	SSL, Kerberos
Служба часу	—	—

Слід зазначити, що модель DCOM, на відміну від CORBA, не є закінченою розподіленою системою, оскільки потребує застосування зовнішніх служб, зокрема служби іменування, обов’язкової для розподіленої системи. Іменування, наприклад в ОС Windows, підтримується за допомогою служби Active Directory, яка містить набір серверів каталогів.

Проте Active Directory не є частиною DCOM, тому у разі активації прикладного програмного забезпечення, побудованого за технологією DCOM, в середовищі UNIX воно не може використовувати службу іменування UNIX, що є обмеженням, яке стосується переносимості між різними операційними системами та перешкоджає міжопераційній сумісності.

4.4. Global Object-Based Environment

Глобальне об’єктне середовище (*GLobal Object-Based Environment, Globe*) – це експериментальна розподілена система, яка націлена на підтримку дуже великої кількості користувачів і об'єктів, розташованих в Internet, зі збереженням повної прозорості розподілу на відміну від CORBA і DCOM. Більшість відомих розподілених систем об'єктів створювали в першу чергу для роботи в локальних мережах, тому вони містять деякі обмеження та недоліки.

У розподіленій системі об'єктів Globe особливу роль відіграє масштабованість. Структуру Globe проектували виходячи з вимог щодо побудови крупних глобальних систем, здатних підтримувати значну

кількість територіально-розподілених користувачів і об'єктів. Основним елементом за такого підходу є метод перегляду об’єктів. Аналогічно іншим системам об'єкти в Globe розглядають як інкапсуляцію сутностей та операцій над ними.

Важливою відмінністю Globe від других масштабованих систем є те, що об'єкти можуть також інкапсулювати реалізацію правил розподілу стану об'єктів декількома машинами, тобто кожний об'єкт визначає, яким чином його стан розміщатиметься репліками. Кожний об'єкт керує також й іншими своїми правилами.

Об'єкти в Globe визначають, як, коли та куди може переміщуватися їх стан, вирішують, чи слід робити репліки їх стану і, якщо так, як саме відбуватиметься реплікація, можуть також встановлювати власні правила захисту і власну реалізацію. Розглянемо яким чином досягається така інкапсуляція.

На відміну від більшості інших розподілених систем об'єктів, Globe не підтримує модель віддалених об'єктів. Розподілені об'єкти, які розділяють стан (*distributed shared objects*) в Globe здатні зберігатися у фізично розділеному стані, коли стан об'єктів може бути розділений і реплікований між декількома процесами.

На рис. 4.8 показано об’єкт, розділений між чотирма процесами, кожен з яких виконується на окремій машині.

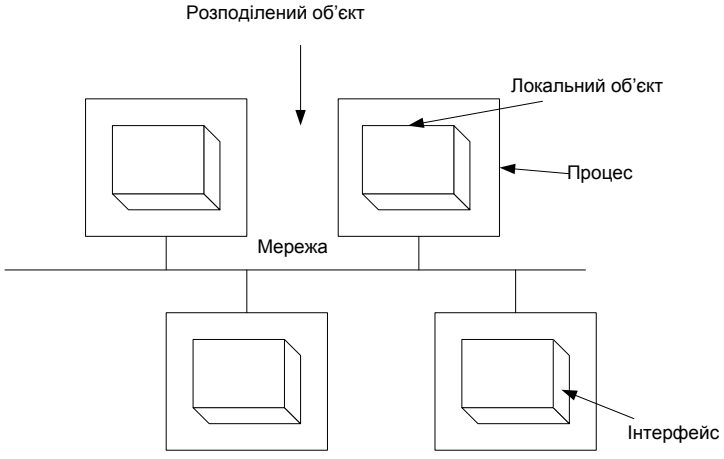


Рис. 4.8. Організація розподіленого об’єкта з поділюваним станом

Процес, пов’язаний з розподіленим об'єктом із поділюваним станом, отримує локальну реалізацію інтерфейсів цього об'єкту. Цю локальну реалізацію називають **локальним представленням** (*local representative*) або **локальним об’єктом** (*local object*). Локальний об'єкт має стан повністю прозорий для процесу, який його містить. Реалізація об'єкту прихована за інтерфейсами, які пропонуються процесу.

Кожний локальний об'єкт в Globe реалізує стандартний інтерфейс об'єкта *Soinf,* аналогічний інтерфейсу *Idnknown* в DCOM, зокрема якщо в DCOM виконують звертання до методу *Queryinterface*, то через інтерфейс об’єкта *Soinf* викликають метод *getInf*, який отримує як вихідні дані ідентифікатор інтерфейсу і повертає вказівник на цей інтерфейс, дозволяючи другим процесам дістати доступ до його реалізації в об'єкті. Локальні об'єкти Globe і DCOM є дуже подібними. У обох технологіях, наприклад, передбачається, що кожному локальному об'єкту співставлено відповідний об'єкт класу, який може створювати нові локальні об'єкти.

Локальні об'єкти реалізують бінарні інтерфейси, які містять в основному таблиці вказівників на функції. Інтерфейси описують на мові визначення інтерфейсів, яка з незначними власними відмінностями аналогічна мовам, використовуваним в CORBA і DCOM.

Розрізняють два різновиди локальних об'єктів Globe: **примітивний локальний об’єкт** (*primitive local object*), який не містить інших локальних об'єктів та **складний локальний об'єкт** (*composite local object*), який створено з декількох (можливо також складних) локальних об'єктів. Для підтримки функціонування складних об'єктів таблиця інтерфейсів локальних об'єктів Globe містить пари вказівників (стан, метод). Кожен вказівник на стан відповідає даним, які належать одному конкретному локальному об'єкту. Інтерфейс

примітивного локального об'єкта визначає всі вказівники на стан, який містить одні й ті ж дані, що відносяться до стану самого об'єкту. Інтерфейс складних об'єктів визначає вказівники на стан, який об’єднує стани різних об'єктів, які включає цей складний об'єкт.

Складні об'єкти використовують для побудови локальних об'єктів, необхідних для реалізації розподілених об'єктів із поділюваним станом, наприклад локальний об'єкт містить як мінімум чотири підоб'єкти (рис. 4.9).

Рис. 4.9. Узагальнена структура локального об’єкту для розподілених об’єктів Globe з поділюваним станом

Підоб’єкт семантики (*semantics subobject*) реалізує функціональність розподіленого об'єкта з поділюваним станом, наприклад його використано для розробки розподілених web-сайтів, в яких колекцію логічно зв'язаних web-сторінок, логотипів, зображень та інших елементів було об'єднано в один документ. Такий *GlobeDoc* документ реалізується локально підоб’єктом семантики наданням інтерфейсів, представлених у табл. 4.3. Кожний файл, який використано як частину web-документа, вважають елементом об'єкту *GlobeDoc*. Застосовуючи інтерфейс документа, який містить метод, що повертає список всіх елементів, можна додавати або видаляти такі елементи. Передбачають, що web-документи можна подати як граф з

коренем. Окремі методи дозволяють визначати та знаходити кореневий елемент, для чого у багатьох web-орієнтованих прикладних програмах і сервісах використано стандартний файл *index.htm*.

Кожний елемент подається як масив байтів. *Інтерфейс вмісту* містить методи отримання поточного вмісту документа і його заміни заданим масивом байтів. *Інтерфейс властивостей* надає методи для маніпулювання з метаданими, пов'язаними з елементом. Метадані елемента подають як пару (атрибут, значення), наприклад, кожний з елементів *GlobeDoc* має власний MIME-тип.

Для того, щоб вносити зміни до елементів паралельно, кожний об'єкт *GlobeDoc* реалізує *інтерфейс блокувань*, який керує паралельним доступом. Оскільки вважають, що більшість web-документів використовує відносно невелика кількість користувачів, механізм блокування є відносно простим. Коли елемент необхідно змінити, то спочатку його *захоплюють* (*check out*), тобто блокують на запис на час модифікації, дозволяючи тільки читати. Після проведення модифікації елемент *звільняють* (*check in*), після чого зміни зберігають.

Підоб'єкт зв'язку (*communication subobject*), який містить декілька методів передачі повідомлень для взаємодії зі встановленням з'єднання і без встановлення з'єднання, використовують для отримання стандартного інтерфейсу з базовою мережею (рис. 4.9). Для більш ефективної взаємодії застосовують досконаліші підоб'єкти зв'язку, які реалізовують інтерфейси групового розсилання. Деякі підоб'єкти зв'язку можуть використовуватися для організації надійного зв'язку, тоді як інші можуть забезпечити лише ненадійний зв'язок.

Дуже важливим майже для всіх розподілених об'єктів із поділюваним станом є *підоб'єкт реплікації* (*replication subobject*), який реалізує обрану стратегію розподілу об'єкта та анологічно підоб'єкту зв'язку має стандартізований інтерфейс. Підоб'єкт реплікації відповідає за те, як саме виконуватиметься метод, поданий підоб'єктом семантики.

Так підоб'єкт реплікації, який реалізовує активну реплікацію, має бути впевненим, що всі звертання до методів виконуються на кожній репліці в одному порядку. В такому разі цей підоб'єкт зможе працювати разом з підоб'єктом реплікації іншого локального об'єкта, який є тим самим розподіленим об'єктом із поділюваним станом.

Підоб'єкт керування (*control subobject*) відіграє роль прошарку між призначеним для користувача інтерфейсом підоб'єкта семантики і стандартизованим інтерфейсом підоб'єкту реплікації. Крім того, він відповідає за експорт інтерфейсів підоб'єкта семантики у процеси, які працюють із розподіленим об'єктом із поділюваним станом. Перед тим, як передати підоб'єкт реплікації, підоб'єкт керування виконує маршалінг усіх звертань до методів, які відправляють такі процеси.

Підоб'єкт реплікації може дозволити підоб'єкту керування самому обробити запит і повернути результат процесу, який звернувся до об'єкта. Так само можна передавати підоб'єкту керування запити віддалених процесів. У такому разі буде виконано демаршалінг такого запиту, після чого підоб'єкт керування виконає його і передасть результати назад підоб'єкту реплікації.

4.4.1. Прив'язка процесу до об'єкта

На відміну від CORBA і DCOM, система Globe не має ні сховища інтерфейсів, ні аналога сховища реалізацій, що частково пов'язано з моделлю об'єктів, яку використано у Globe. Зокрема, коли відбувається прив'язка процесу до об'єкта, процес має завантажити у свій адресний простір конкретного локального об'єкта, який відповідає розподіленому об'єкту з поділюваним станом, до якого виконується прив'язка.

Прив'язка процесу до об'єкта виконується за п'ять кроків (рис. 4.10), при цьому кожен крок, окрім останнього, повертає інформацію, необхідну для наступного кроку. Отже, якщо отримано інформацію для

чергового кроку прив'язки, то цей крок можна починати здійснювати, що підвищує ефективність прив'язки в цілому.

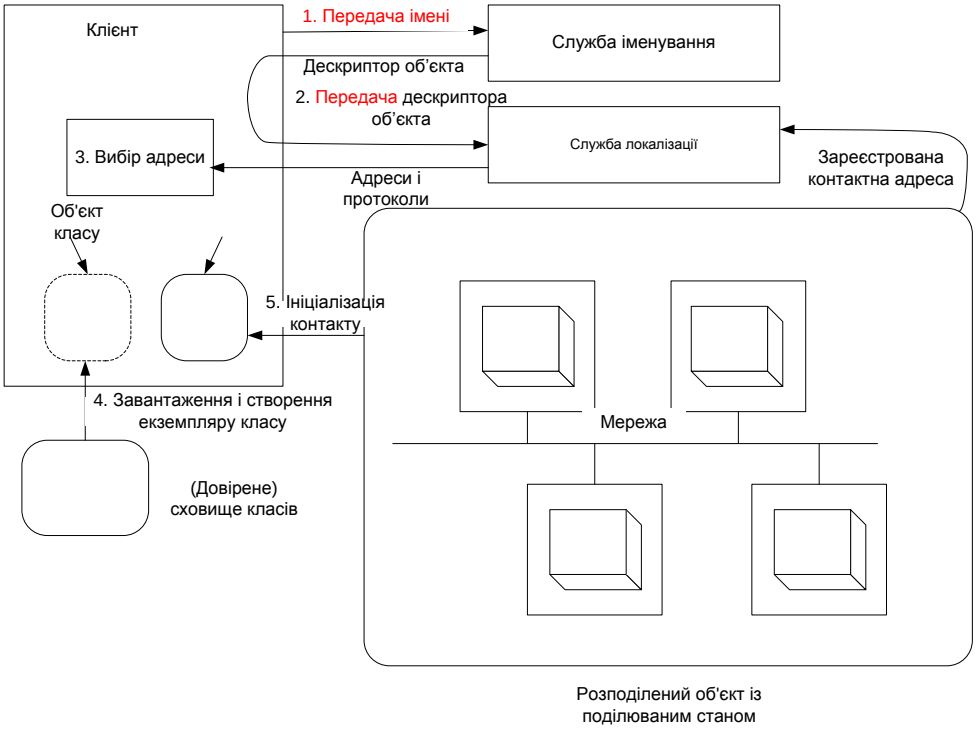


Рис. 4.10. Прив'язка процесу до об'єкта у Globe

Прив'язка починається з пересилання осмисленого (з позицій людини) імені службі іменування (рис. 4.10 - крок 1). Globe містить службу іменування на основі DNS, яка повертає глобально унікальне ім'я і не залежний від місця розташування *дескриптор об'єкта* (*object handle*) (рис. 4.10 - крок 2). Дескриптор об'єкта використовується у Globe як глобальне посилання на об'єкт, що дозволяє виконувати пошук об'єкта за допомогою служби локалізації Globe. Відзначимо, що Globe відокремлює службу іменування від служби локалізації. Перевага такого поділу полягає в тому, що можна змінювати імена й адреси, не впливаючи на відображення імен в адреси.

Служба локалізації Globe повертає набір *контактних адрес* (*contact address*) цього об'єкта, що точно визначають, де та яким чином можна знайти об'єкт, який може мати декілька контактних адрес, наприклад, у зв'язку з реплікацією або підтримкою декількох комунікаційних протоколів. Контактна адреса нагадує *міжопераційне*

посилання на об’єкт (*IOR*) у CORBA. Після отримання контактної адреси можна розраховувати на те, що у процесу є вся необхідна для прив'язки до об'єкта інформація.

Оскільки служба локалізації може повернути набір контактних адрес для одного об’єкта, процес має перш за все обрати один з них (крок 3). Вибір може бути довільним або спиратися на певні критерії, зокрема на відстань до адреси або передбачувану якість обслуговування в разі прив'язки до цієї адреси.

Будь-яка контактна адреса містить інформацію про те, що необхідно для створення локальної реалізації процесу, щоб він міг працювати з об'єктом, зокрема він точно визначає локальний об'єкт, який процесу слід завантажити. Наприклад, контактна адреса може містити URL-адресу файлу, що містить об’єкти класу, які повинен завантажити процес. Подібний підхід нагадує завантаження класів у Java. Завантаження локального об'єкта зі сховища (довіреного) класів і створення екземпляра ілюструє крок 4. Зазначимо, що сховище класів може бути файловою системою віддаленого сайту з доступом за FTP або за іншим протоколом обміну файлами.

Сховище класів подібне до сховища реалізацій CORBA в тому сенсі, що обидва вони містять реалізації об'єктів, проте у Globe сховище класів більше відповідає традиційній схемі зберігання, в якій вибирається код, тоді як у CORBA- це процес, якому можна відправити запит на створення об’єкта на сервері об’єкта.

Крок 5 полягає в ініціалізації локального об'єкта й організації через цей об'єкт роботи з іншими локальними об'єктами, які є частиною розподіленого об'єкту з поділюваним станом.

4.4.2. Служби Globe

На відміну від CORBA і DCOM, Globe містить порівняно небагато служб, а саме лише ті служби, які не надає сам об'єкт. Це просто пояснити, розглядаючи деякі служби, які аналогічно реалізовано у Globe та містяться в CORBA і DCOM. Служба колекцій CORBA використовується для об'єднання об'єктів у списки, черги тощо, та зазвичай може бути реалізована за допомогою об'єкта, станом якого є набір посилань на об'єкти.

Служба паралельного доступу в CORBA визначена як колекції інтерфейсів, які використовують для доступу різних типів блокувань. У DCOM керування паралельним доступом здійснюється частково службою транзакцій, а частково так само, як сервери об'єктів обслуговують потоки виконання. У Globe керування паралельним доступом виконує кожний об’єкт самостійно, тобто розподілений об'єкт з поділюваним станом, якому необхідно запобігти паралельному доступу до свого стану, має надати певний спеціальний інтерфейс для блокування. Жодного стандартного інтерфейсу для керування паралельним доступом у Globe не існує.

Як і в колекціях, у транзакціях також відбувається об'єднання об'єктів. У Globe транзакції зазвичай реалізуються за допомогою віддаленого об'єкта, що відіграє роль менеджера транзакцій. Інтерфейси для подібних об'єктів не стандартизовано.

У Globe немає служби подій, об'єкти завжди пасивні в тому сенсі, що у них немає власного потоку виконання, тобто об'єкт не в змозі повідомити клієнта про подію, що сталася. Для реалізації служби подій у Globe є придатною модель *отримання* (*pull*) CORBA, коли події можна об'єднати в один об'єкт, як у класі подій DCOM. До нині події у Globe не використовувалися.

Служба зовнішніх зв'язків, або служба маршалінгу, зазвичай реалізується для кожного об'єкта окремо. У Globe більшість об'єктів мають реалізовувати власні процедури маршалінгу, за допомогою яких вони отримують можливість передавати свій стан на інші машини.

Служба життєвого циклу надає засоби створення, видалення, копіювання і переміщення об'єктів, але у Globe об'єкт сам себе створити не може, а всі інші операції реалізуються самим об'єктом, тому що спеціальних засобів для видалення, копіювання або переміщення об'єктів у Globe немає. Створюються об'єкти зазвичай через пряме звертання до сервера об'єкта з вимогою створити екземпляр об'єкта. Подібний підхід дещо схожий на спосіб, який застосовується в DCOM.

Ліцензування у Globe не підтримується, але може бути реалізовано звичайним способом окремо для кожного об'єкта, який цього потребує.

Службу іменування неможливо реалізувати за допомогою іменованого об'єкта, оскільки її використовують для пошуку цього об'єкта, тобто доступ до служби іменування за визначенням вимагається раніше, ніж доступ до іменованих об’єктів. Ця вимога обумовила неможливість реалізувати службу іменування з використанням об'єктів, як у CORBA. Отже у Globe є виділені служби іменування, властивостей і торгівлі, які реалізовано окремо від об'єктів, посиланнями на які вони маніпулюють.

Більшість розподілених систем має виділену службу збереження (тривалого зберігання) у формі файлової системи або бази даних, але Globe на відміну від них такої служби немає. Замість цього збереження розглядається як властивість об'єкта, а тому має реалізовуватися самим об'єктом. Як зберігати вирішує об'єкт, але ним у Globe для тривалого зберігання свого стану передусім застосовуються сервери об'єктів, хоча багато реалізацій об'єктів використовують також локальну файлову систему.

Захист у Globe частково реалізує кожний об'єкт окремо, а частково - локальні та глобальні служби захисту. Наприклад, хоча об'єкт сам може контролювати велику частину аспектів, пов'язаних зі звертанням до методів, питання отримання і сертифікації ключів зазвичай виділяють в окрему службу.

Основна відмінність між Globe та іншими розподіленими системами об'єктів стає очевидною під час розгляду питань реплікації, яа якої у Globe об'єкти самі визначають, як потрібно реплікувати їх стан та виключно самі це здійснюють, натомість у таких системах, як CORBA, зазвичай застосовуються спеціальні сервери реплікації, які керують реплікацією груп об'єктів.

Відмовостійкість у Globe також підтримується кожним об'єктом, хоча для успішного маскування помилок необхідна певна підтримка з боку серверів відмовостійких об'єктів.

Більш детальну інформацію щодо побудови системи Globe наведено у [1,2.4,5].

4.5. Порівняльна характеристика технологій CORBA, DCOM і Globe

CORBA, DCOM і Globe – три різні системи об’єктів зі своїми перевагами і недоліками, які розробляли з використанням різних підходів. CORBA є фактичним результатом спроб створити стандартну платформу проміжного рівня, на якій могли б спільно працювати прикладні програми від різних виробників. Сотні учасників різних комітетів прагнули дійти згоди щодо того, як мають виглядати інтерфейси компонентів CORBA.

Основною метою розробки DCOM було розширення функціональних можливостей з одночасним збереженням сумісності з наявними попередніми версіями системи Windows.

Система Globe є типовим зразком дослідницької роботи, яку розроблено невеликим колективом, що віддавав перевагу простоті й функціональності. Проте Globe – незавершена система і багато її аспектів вимагають доопрацювання. Основна мета розробки Globe – забезпечення масштабованості.

Головні відмінності об'єктних моделей, які підтримують три системи, такі: у CORBA і DCOM використовується модель віддалених об'єктів, причому CORBA пропонує гнучку і несуперечливу об'єктну модель, яка забезпечує високий ступінь прозорості операцій визначення місцезнаходження і доступу до об'єктів, через що клієнт майже не розрізняє відмінностей між локальним і віддаленим об'єктами; об'єкти мають стан, можуть бути глобально ідентифікованими, а посилання легко передаються від клієнта до клієнта і від машини до машини. Крім того, об’єкти можуть бути як нерезидентними, так і такими, які підлягають зберіганню.

Модель DCOM значно простіша за модель CORBA через те, що об'єкти в DCOM є нерезидентними, не мають глобальних ідентифікаторів, а іноді не мають і стану, що порушує багато принципів, у основу яких покладено технології розподілених об'єктів.

Система Globe підтримує правильні об'єкти, особливісттю об'єктної моделі яких є те, що об'єкти у Globe можуть бути реально реплікованими та розподіленими по декількох машинах. Більш того, об'єкт визначає, як можна виконувати розподіл і реплікацію його стану, тобто інкапсулює власні правила розподілу і реалізації, а також правила і реалізації захисту, обробки помилок тощо.

Порівнюючи служби цих систем, слід відзначити суттєві відмінності. Система CORBA містить набір служб, широкий настільки, що функціональність окремих служб часто перетинається. Натомість DCOM пропонує поєднання власних служб зі службами оточення, зокрема службами іменування і каталогів. У Globe реалізовано лише

мінімум служб – проста служба іменування і розширена служба локалізації, що відповідає логіці побудови Globe, проте для перетворення її на справді працездатну розподілену систему загального призначення необхідно мати множину додаткових служб.

Інтерфейси є ще однією істотною відмінністю, за якою розрізняють три системи. CORBA використовує стандартну мову IDL, яка призначена для визначення інтерфейсів, що перетворюються потім на тексти програм обраною мовою програмування. З погляду стійкості й міжопераційної взаємодії таке відображення на мову програмування стандартизовано. Основна перевага такого підходу полягає в тому, що CORBA не залежить від коду, що генерується компіляторами, а визначає міжопераційну взаємодію на рівні мови програмування.

На відміну від CORBA системи DCOM і Globe підтримують бінарні інтерфейси, коли інтерфейси об'єктів визначаються незалежно від мови програмування. У DCOM і Globe переваги бінарних інтерфейсів можна побачити на прикладі численних прикладних програм і сервісів, написаних на таких мовах, як C, Java і Visual Basic.

Переваги застосування технології DCOM такі:

- незалежність від мови програмування;
- можливість динамічного і статичного виклику об’єкта;
- динамічне знаходження об'єктів;
- масштабованість;
- використання відкритого стандарту (контроль з боку TOG).

Серед наявних недолів технології DCOM можна назвати:

- складність реалізації;
- залежність від платформи;
- немає іменування через URL;
- немає перевірки безпеки на рівні виконання ActiveX компонент;
- DCOM - вирішення проблеми розподілених об'єктних систем, яке лише придатне для Microsoft-орієнтованих середовищ.

Якщо потрібно працювати з архітектурою, яка відрізняється від рішень компанії Microsoft, DCOM стає неефективним і ресурсомістким у використанні.

Переваги використання технології CORBA такі:

- платформна незалежність;
- незалежність від мови програмування;
- можливість динамічного і статичного виклику об’єкта;
- динамічне виявлення об'єктів;
- масштабованість;
- можливість використання CORBA-сервісу;
- широка індустріальна підтримка;
- коло виробників продуктів, які підтримують цю технологію, значно ширше для DCOM.

У якості недоліків технології CORBA слід відзначити:

- неможливо передати параметри «за значенням»;
- відсутнє динамічне завантаження компонент – перехідників, які здійснюють тільки перехід за посиланням до потрібного елементу даних або компоненту виконання процесу;
- немає іменування через URL;
- комерційні реалізації CORBA дуже дорогі;
- для використання платформи потрібний дуже високий рівень знань;
- недоліки розробки протоколу інтероперабельності CORBA унеможливлюють побудову високопродуктивної *служби поширення подій* (*event distribution service*);
- у використовуваному в CORBA способі кодування аргументів і результатів викликів є значна надмірність, але у протоколі не підтримується стиснення, що зумовлює низьку продуктивність під час роботи в територіально розподілених мережах;

− у специфікації майже не враховано можливість застосування *багатопотоковості* (*threading*), тому багатопотокове прикладне програмне забезпечення не можна переносити (хоча багатопотоковість дуже важлива для комерційних програм);

− у CORBA не підтримується асинхронна диспетчеризація на боці сервера;

− немає мовного відображення для C# і Visual Basic, а також повністю не підтримується .NET.

4.6. Web-сервіси

Технологію Web-сервісу створено для підтримки сервіс-орієнтованої архітектури (*SOA*).

«Орієнтація на сервіси» часто протиставляється підходу «дрібноструктурних» (*fine-grain*) розподілених об'єктів, який вважають визначальною характеристикою систем, що грунтуються на CORBA.

Застосування Web-сервісу не гарантує побудову хорошої розподіленої системи, а використання CORBA не обов'язково призводить до створення поганої системи. Web-сервіси не надають системі можливість ефективно подолати обмеження пропускної здатності й затримки.

Web-сервіси не запобігають труднощам, що виникають через часткові відмови і залежності під час функціонування розподілених систем, які можуть бути тимчасово недоступними.

Web-сервіси (Web-служби) – це технологія, яка дозволяє прикладним програмам взаємодіяти незалежно від платформи, на якій їх розгорнуто, а також від мови програмування, якою їх написано.

Web-сервіс - це програмний інтерфейс, який описує набір операцій, що можуть бути викликані віддалено мережею за допомогою стандартизованих повідомлень XML. Для опису операції або

викликуваних даних використовуються протоколи, які грунтуються на мові XML.

Група Web-сервісів, які взаємодіють таким чином, визначає спеціально організовану прикладну програму або Web-сервіс, який працює в межах сервіс-орієнтованої архітектури (*SOA*).

Основною проблемою, яку вирішують Web-сервіси, є інтеграція даних і прикладного програмного забезпечення, а також перетворення технічних функцій у бізнес-орієнтовані автоматизовані задачі. Ці два аспекти дозволяють різним компаніям взаємодіяти на рівні процесів або прикладних сервісів з їх партнерами, маючи при цьому можливість динамічно адаптуватися до нових ситуацій або працювати з різними партнерами залежно від поставлених вимог.

В індустрії програмного забезпечення прогресують тенденції інтеграції прикладних програм, що функціонують на різних операційних системах, мовах програмування та апаратних платформах, тенденції усунення жорсткої прив'язки прикладних програм одна до іншої назвою функції і її параметрами. У розподілених системах, які розроблено з використанням технологій, створених раніше за технології Web-сервісу, здебільшого використовувався фіксований малогнучкий програмний інтерфейс, з обмеженими можливостями щодо його адаптації до змінюваних вимог середовища або потреб різних систем.

Web-сервіси використовують мову XML, за допомогою якої можна описати будь-які дані незалежним від платформи способом для обміну інформацією між системами, що, у свою чергу, зумовлює слабку зв'язність прикладного програмного забезпечення. Крім того, Web-сервіси можуть функціонувати на більш високому рівні абстракції, аналізуючи, модифікуючи чи обробляючи типи даних динамічно за вимогою. З технічного погляду, Web-сервіси спроможні

обробляти дані значно легше, надаючи можливість компонентам програмного забезпечення взаємодіяти між собою більш відкрито.

На найвищому концептуальному рівні Web-сервіси - одиниці прикладного програмного забезпечення, кожна з яких виконує певне функціональне завдання. На наступному більш вищому рівні такі завдання можна об'єднати в бізнес-орієнтовані задачі для виконання визначених бізнес операцій, дозволяючи технічно непідготовленим фахівцям в галузі інформаційних технологій розглядати прикладні програми як обробники бізнес завдань у межах потоку робіт прикладного програмного середовища Web-сервісу.

Після розробки Web-сервісів архітектори бізнес-процесів можуть об'єднувати їх для вирішення конкретних бізнес-завдань. Web-сервіси допомагають у межах компанії організувати колективну роботу фахівців з різних галузей знань.

Універсально описані інтерфейси і добре спроектовані завдання дозволяють здійснити їх повторне застосування для різних сфер діяльності, а отже і повторне використання прикладного програмного забезпечення.

4.6.1. Сервіс-орієнтована архітектура

Сервіс-орієнтована архітектура (*SOA*) - це принципи побудови й використання розподіленої функціональності (*capabilities*), яка може керуватися власними розподіленими центрами, розташованими в одних із нею зонах.

Ключові ознаки опису SOA-парадигми такі: видимість, взаємодія й ефект, що подано умовно на рис.4.11.

Видимість (*visibility*) – здатність надати необхідну функціональність за запитом і визначити стороні, яка надсилає запит, який сервіс цей запит може задовольнити, що досягається за рахунок

опису функцій і технічних вимог, пов'язаних з цим обмежень і політики взаємодії, а також механізмів для доступу та одержання відповіді. Описи мають бути форматованими (або мати можливість бути відформатованими) з використанням загальнодоступного і зрозумілого *синтаксису й семантики* (*semantics*).

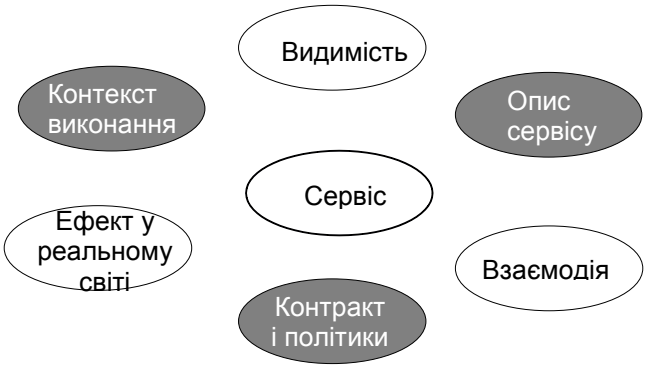


Рис. 4.11. Ключові ознаки опису SOA-парадигми для еталонної моделі сервіс-орієнтованої архітектури

Видимість дозволяє визначити напрям взаємодії (який запит яким сервісом обробляється), *взаємодія* (*interaction*) – процес виконання необхідної функціональності, тобто отримання результату за запитом, наприклад, у процесі проміжного обміну повідомленнями взаємодія відбувається як послідовність обмінів інформацією й викликів дій, які співставлено з певним *контекстом виконання* (*execution context*) – множиною технічних і бізнес елементів, які зв'язують запити та сервіси, що, у свою чергу, складаються із цих елементів. Усе це дозволяє постачальникам і споживачам сервісів взаємодіяти й дозволяє прийняти рішення щодо використання різних політик і контрактів.

Мета використання довільних запитів в глобальному середовищі, які здатні самостійно знайти необхідний для їх обробки сервіс, - одержання одного (або більше) *ефекту в реальному світі* (*real world effects*), тобто потрібного користувачеві результату обробки запиту, зокрема у вигляді одержання інформації або зміни стану об'єктів (відомих або невідомих), залучених до взаємодії.

Розмежуємо загальні (*public*) і часткові (*private*) дії: часткові дії за своєю основою недоступні іншим частинам, які їх не бачать, натомість результати загальних дій виражаються у змінюванні поділюваного у поточному контексті виконання *стану* (*state*), при цьому можливим є поділ і в інших контекстах. Ефекти в реальному світі за таких умов є вираженими в часі змінами цього *поділюваного стану* (*shared state*).

На рівні взаємодії опис ефектів у реальному світі вказує, якими мають бути очікувані результати обробки певного запиту в глобальному середовищі. Важливим принципом, використаним у SOA є те, що подаючи запит користувач не має знати деталей реалізації сервісів.

Такий опис SOA найчастіше розглядають з погляду основного поняття: *сервіс* (*service*), який у словниках визначають як «виконання роботи (функції) одним для іншого». Не зважаючи на це, сервіс, як термін у широкому значенні, охоплює такі вимоги:

- можливість виконання роботи для будь-кого за його запитом;
- специфікація роботи, яку сервіс здатний виконати;
- пропозиція виконати роботу для будь-кого за його запитом.

В SOA, сервіси є механізмами, за допомогою яких запити від будь-якого користувача поєднуються з функціональними можливостями сервісів, які розміщено будь-де в середовищі Інтернет.

Концепції видимості, взаємодії й ефекту застосовуються так само і до сервісів. Видимість забезпечує *опис сервісу* (*service description*), що містить інформацію, необхідну для взаємодії з ним, зокрема такі параметри як вхідна та вихідна інформація, співставлення семантики, а також вказує, що роботу сервісу буде завершено у разі залучення сервісу у відповідності до умов його використання.

Об’єкти (люди й організації), пропонуючи певні сервіси, відіграють роль *постачальників сервісу* (*service providers*), а ті, хто їх потребує, - *споживачів сервісу* (*service consumers*). Опис сервісу

дозволяє майбутнім споживачам прийняти рішення щодо відповідності сервісу їх поточним потребам, а також визначитись постачальникам сервісу відповідності споживачів вимогам його надання. Постачальників і споживачів сервісів іноді називають *сервісними учасниками* (*service participants*), а такий зв'язок в глобальному середовищі між ними - «слабким зв'язком».

SOA зазвичай розглядають як інструмент, що використовує Web-сервіси, які можуть бути доступними для аналізу їх відповідності певному запиту, підтримують взаємодію і дозволяють отримати необхідний результат певним способом, визначеним в еталонній моделі. Проте архітектури й технології, які грунтуються на Web-сервісах, є специфічними й залежать від вимог взаємодіючих сторін. На відміну від концепцій еталонної моделі, що застосовують до окремих систем, Web-сервіси ззалежать від програмно-технічних рішень платформи надання послуг, які виступають частиною основної еталонної моделі.

Опис сервісу. Одна з ознак сервіс-орієнтованої архітектури - велика кількість зв'язаних документів й описів. Опис сервісу містить інформацію, потрібну для того, щоб використати сервіс. Здебільшого не існує жодного «правильного» опису, а лине необхідні елементи, які залежать від контексту й потреби сторін, що використають взаємодійний об'єкт.

Опис сервіс включає певні елементи, які є обов’язковою частиною будь-якого опису прикладного програмного забезпечення, а саме інформаційну модель, та інші елементи, перелік яких може варіюватися, зокрема доступні функції й політику використання сервісу.

Мета використання опису: спростити процес взаємодії сервісів й видимість між його учасниками, особливо коли вони є територіально-розподіленими. Існування описів дає можливість потенційним

учасникам створювати системи, використовуючи необхідні сервіси, і досить просто їх отримувати. Наприклад, за допомогою описів учасники виокремлюють потрібні дані для взаємодії сервісів, зокрема запити кінцевих користувачів на певні інформаційні послуги, інформацію від постачальників сервісів щодо політики доступу до сервісу, який надаватиме ці послуги, та застосування його специфічної функціональності. Крім того, описи можуть використовуватися для підтримки й керування сервісами, як з боку постачальника сервісу, так і з боку споживача.

Найкраще рекомендується подавати опис вимог до сервісу, враховуючи стандарт і затверджений формат, який спрощує використання загальних інструментів розробки процесів (наприклад, механізмів знаходження сервісів, які застосовують для порівняння описів вимог до сервісів з описами самих сервісів).

Концепція SOA підтримує використання сервісу, коли споживачеві не потрібно знати деталей сервісної взаємодії, опис сервісу застосовує тільки ту інформацію, яку запит споживача надає через опис вимог до сервісу, щоб визначитись щодо використання або невикористання сервісу. Зокрема, споживач сервісу має потребу в одержанні такої інформації:

- що сервіс існує та доступний (*reachability*);
- що сервіс виконує деяку функцію або набір функцій;
- що сервіс функціонує з дозволу певного набору обмежень і політик;
- що сервіс (до деяких явних або неявних меж) взаємодіятиме на підставі політик, узгоджених зі споживачем сервісу;
- як взаємодіяти із сервісом для досягнення необхідних цілей, включаючи формат і вміст інформаційного потоку між сервісом і споживачем й очікуваною послідовністю обмінюваної інформації.

Кожний із цих пунктів має бути в описі сервісу, деталі можуть бути доступними через посилання на зовнішні джерела, які не вбудовують явно, що дозволяє повторно використати стандартні описи, зокрема, для функціональності або політики.

Політика й контракти. *Політика* (*policy*) являє собою деякі обмеження або умови використання, прикладну програму або описи об'єкта його власником для будь-якого учасника, натомість *контракт* (*contract*) - договір двох або більше сторін. Як і політика, контракти стосуються умов використання сервісу, можуть відокремлювати передбачувані ефекти в реальному світі від використання сервісу. Еталонну модель зосереджено передусім на концепції політики та обмежень, а також умов їх застосовування до сервісів.

Політика сервісу. Концептуально розрізняють три аспекти політик: *політика тверджень* (*assertion*), *власник політики* (*policy owner*) (іноді сам сервіс є власником своєї політики) і *політика правозастосування* (*policy enforcement*), яка найчастіше застосовується до реалізації сервісу, тобто вона є зв'язком між сервісом і контекстом його виконання. Наприклад, «Всі повідомлення зашифровані» є твердженням щодо виду повідомлень і є відносним: може бути істиною або неправдою, зокрема залежно від того, чи шифрується трафік чи ні.

Політика завжди подає погляди учасників. Твердження стають політикою учасника, коли він приймає твердження як свою політику, наприклад, якщо споживач сервісу декларує: «Всі повідомлення мають бути зашифровані», то це відображається та затверджується в його політиці незалежно від інших угод з боку постачальника сервісу.

Виконання деяких умов політики може вимагатися певним учасником взаємодії, але технології, які реалізують це, залежать від їх природи. Концептуально політика умов застосування сервісу означає гарантованість того, що політику тверджень погоджено з реальною можливістю застосування сервісу. Це може означати запобігання

виконанню неприпустимих дій або виключення певних станів, а також виконання компенсувальни дій у разі виявлення порушень умов узгодженої політики. Обмеження, які не можуть бути примусово встановленими, не є політикою, а можуть бути побажаннями.

Політики потенційно застосовуютьдо багатьох аспектів SOA, зокрема до безпеки, конфіденційності, керованості, якості обслуговування. Крім орієнтованих на інфраструктуру політик, учасники можуть також встановлювати бізнес-орієнтовані політики – робочий час, політики повернення тощо.

Політика тверджень має бути написана у вигляді, який можуть зрозуміти та обробити сторони, на які спрямована політика. Політики можуть автоматично інтерпретуватися, залежати від мети й здатності застосувати політики, а також від того, як умови політики стосуються використаного або невикористаного певного сервісу.

Зв'язок між учасниками взаємодії сервісів й співставленими з ними політиками застосування сервісів встановлюється за допомогою опису сервісу, в якому передбачено посилання на пов'язані із сервісом політики

Контракт сервісу. Оскільки політика пов’язана з поглядом окремого учасника, то контракт є угодою між двома або більше учасниками. Як і політики, контракти можуть містити широке коло аспектів взаємодії сервісів: якість контракту сервісу, інтерфейс і хореографію контракту, а також комерційні аспекти.

Отже сервісний контракт є узгодженим поміркованим твердженням, що визначає вимоги й очікування двох або більше сторін. На відміну від політики, яка вимогає дотримання певних умов, що зазвичай належить власникові політики, погодження вимог може відбуватися за рахунок дискусії між сторонами контракту. Результати таких дискусій можуть розглядатися на більш високих рівнях.

Як і політики, контракти можуть дозволяти автоматичну інтерпретацію того, наприклад, де вони використовуються, тобто для систематизації результатів взаємодії сервісів, надання їх у вигляді, зручному для машинної обробки або для перегляду кінцевим користувачем.

Оскільки контракт став результатом згоди залучених сторін, його вважають *процесом,* який співставлено з погодженою дією. Навіть у разі неявного погодження контракту наявна логічно погоджена дія, яка відповідає умовам контракту, навіть якщо вона не відкрита дія угоди. Контракт може доставлятися механізмом, який прямо не є частиною SOA, такий механізм є незв'язаним процесом, альтернативно, контракт, який може бути доставлений механізмом у момент взаємодії сервісів, використовує внутрішній процес.

Контекст виконання (*execution context*) взаємодії сервісів – це множина інфраструктурних елементів, об'єктів процесу, політик тверджень та угод, визначених як частина взаємодії сервісів, а також різноманітних способів зв’язку довільних запитів кінцевих користувачів та наявних в глобальній мережі сервісів.

Опис сервісу містить інформацію, яка може охоплювати передбачувані протоколи, семантики, політики та інші умови і твердження, які описують, як сервіс має використовуватися. Учасники (постачальники, споживачі й треті сторони) повинні дотримуватися й знати про *несуперечливу* множину угод, які подано у контексті виконання, для правильної взаємодії із сервісом, тобто реалізацію описаних ефектів у реальному світі.

Щоб споживач і постачальник могли уявляти собі різні точки на карті взаємодій для реального виклику сервісу, шлях, який є контекстом виконання, має явно вказувати їх розташування в мережі Internet. Шляхом між точками (вузлами мережі, де розмішені учасники взаємодії) може бути тимчасове з'єднання (наприклад, слабкий зв'язок

спеціального обміну) або чітка координація (збережений шлях для постійного використання), що дозволяє в майбутньому мати швидкий та простий доступ до сервісу.

Контекст виконання охоплює весь процес взаємодії, включаючи постачальника сервісу, споживача сервісу й загальну інфраструктуру, необхідну для зв'язування об’єктів взаємодії, визначає, наприклад, точку узгодження рішення щодо політик вимог, які висуваються до взаємодії сервісів. Механізм реалізації вимог політики виконання можливо розмістити в обліковому записі конкретного реального контексту виконання. Крім того, у процесі взаємодії можуть приймати участь і треті сторони, наприклад, загальносистемні сервіси, які висувають деякі умови для контексту виконання, що лише підвищують умови й обмеження, і можуть вимагати додаткового обміну інформацією для закінченого контексту виконання.

Контекст виконання також дозволяє нам розрізняти сервіси один від одного. Різні екземпляри того самого сервісу застосовують у різних процесах взаємодії, які відбуваються між наявним постачальником сервісу й різними його споживачами, тобто розподіляються через різні контексти виконання.

Отже, контекст виконання – це контекст, у якому інтерпретуються дані, обмін якими відбувається. Конкретний рядок в описі сервісів має конкретне значення під час їх взаємодії через контекст виконання.

Контекст виконання часто визначається під час взаємодії сервісів. Крім того, багато інфраструктурних елементів, політик й угод, які застосовуються до взаємодії, також можуть змінюватись під час конкретної взаємодії сервісів. Наприклад, якщо у початковій точці взаємодії погоджено рішення сторін щодо шифрування майбутнього з'єднання, то результат контексту виконання також слід змінити, приєднавши необхідну інфраструктуру для підтримки шифрування і продовження взаємодії.

4.6.2. Стандарти для Web-сервісів

У основі Web-сервісу лежать Internet-стандарти, які визначають протоколи, а не способи їх реалізації. Стандарти Web-сервісів розробляють у співпраці такі компанії, як IBM, Microsoft, Ariba та деякі інші, обговорює комітет World Wide Web Consortium (*W3C*).

Web-сервіси грунтуються на трьох основних Web-стандартах:

− SOAP (*Simple Object Access Protocol*) – протокол для надсилання повідомлень за протоколом HTTP та іншими Internet-протоколами;

− WSDL (*Web Services Description Language*) – мова для опису програмних інтерфейсів Web-сервісу;

− UDDI (*Universal Description, Discovery and Integration*) – стандарт для індексації Web-сервісу.

На рис. 4.12 показано, як ці три стандарти взаємодіють, зокрема для пошуку Web-сервісу використовують реєстр UDDI, на мові WSDL виконують опис Web-сервісу, зокрема усі його програмні інтерфейси, на сервері прикладного програмного забезпечення зберігають Web-сервіси, а мовою XML описують інтерфейси взаємодії.

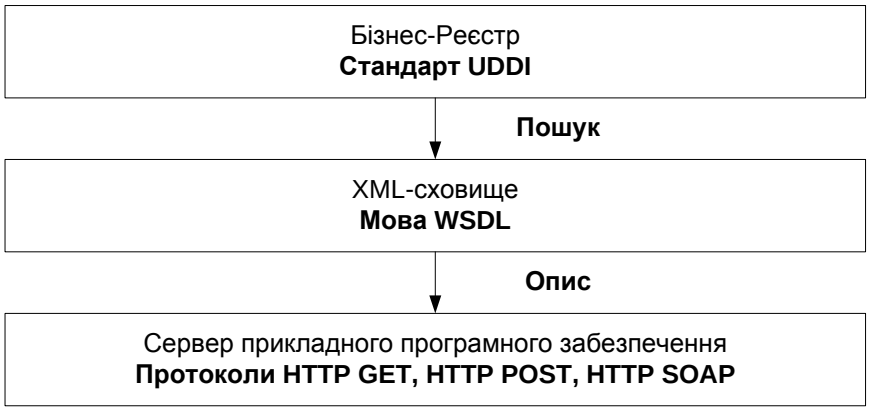


Рис. 4.12. Взаємодія Web-стандартів

Сервери прикладних програм є сховищами Web-сервісів і забезпечують можливість доступу до них через протоколи HTTP GET, HTTP POST і HTTP SOAP.

Наявні Web-сервіси описано в WSDL-документах, які містяться або на сервері прикладного програмного забезпечення, або у спеціальних XML-сховищах. WSDL-документ може посилатися на інші WSDL-документи і документи XSD (XML Schema), в яких описано типи даних, які використовують Web-сервіс. XML-сховищ використовуються для керування WSDL-документами, всередині яких містяться адреса (*URL*) Web-сервісу. Web-сервіси описано і проіндексовано в бізнес-реєстрі, що містить адреси (*URL*) WSDL-документів.

4.6.3. Переваги технології Web-сервісів

Об'єкти можуть описати сервіси, що взаємодіють з іншими сервісами за правилами, описаними на XML, за рахунок чого кожен сервіс транслює та аналізує повідомлення відповідно до своєї локальної реалізації та середовища. Отже, мережні сервіси можуть бути створені з ряду різних об’єктів у разі, якщо вони відповідають правилам, визначеним їх сервіс-орієнтованою архітектурою.

Технологія Web-сервісів дозволяє:

− організовувати взаємодію між різними сервісами на будь-якій платформі, написаними будь-якою мовою програмування;

− перейти від функцій прикладної програми до концепції завдання, що зумовлює аспектно-орієнтовану розробку і потоки робіт. Це дозволяє досягти вищого рівня абстракції програмного забезпечення, за якого воно може бути задіяне користувачами, котрі не є спеціалістами в галузі програмно-технічних засобів, які працюють на рівні бізнес - аналізу;

− враховувати слабкі зв’язки, це означає, що взаємодія між прикладним програмним забезпеченням сервісу не порушується кожного разу, коли змінюється дизайн або реалізація якого-небудь сервісу;

− адаптувати наявні прикладні програми до змінних умов бізнесу і потреб замовника;

− надавати наявному або успадкованому програмному забезпеченню сервісний інтерфейс, не змініючи оригінальних програм і даючи їм можливість повноцінно взаємодіяти в сервісному середовищі;

− додавати інші адміністративні функції або такі функції, які керують обчисленнями, зокрема надійність, підзвітність, безпека, і не залежать від проблемно-орієнтованих функцій, підвищуючи тим самим гнучкість і корисність обчислювального середовища бізнесу.

Основні можливості керування Web-сервісами, які можна конфігурувати в Oracle Application Server, наприклад, такі:

− аудит (*Auditing*) – ведення повної та узгодженої фіксації SOAP-запитів і помилкових ситуацій;

− журналізація (*Logging*) – забезпечення створення журналів на основі контенту із застосуванням *Xpath*, щоб запрошувати вхідні (*inbound*) та вихідні (*outbound*) SOAP-повідомлення;

− надійність (*Reliability*) – конфігурація надійного поширення повідомлень, забезпечення послідовного порядку проходження повідомлень (*message ordering*) і недопущення дублікатів повідомлень відповідно до стандарту *WS-Reliability*;

− безпека (*Security*) – конфігурація аутентифікації, цілісності завдяки використанню цифрових підписів, конфіденційності за допомогою шифрування відповідно до стандарту *WS-Security*;

− основні можливості керування життєвим циклом (*Basic lifecycle management*) – вмикання (*enabling*) і вимикання (*disabling*) сервісів і можливостей керування ними;

− розгортання (*deployment*) – конфігурація стандартних і таких, що переносяться, планів розгортання для Web-сервісу відповідно до стандартів J2EE.

4.6.4. Протоколи Web-сервісів

Web-сервіси використовують сімейство зв'язних протоколів для опису, доставки і взаємодії з сервісами, яке можна розділити на підгрупи, ґрунтуючись на функціях загального використання, зокрема розглянемо підгрупу, що вирішує питання пов'язані з доставкою повідомлень, описом інтерфейсів, адресацією і доставкою сервісів.

Найбільш поширеним є протокол обміну повідомленнями *SOAP*, який кодує повідомлення так, щоб вони могли бути доставлені мережею з використанням транспортного протоколу, зокрема HTTP, IIOP, SMTP.

Мова опису Web-сервісу (*WSDL*) подана як набір XML виразів, які визначають інтерфейс для кожного сервісу.

Друга група протоколів і специфікацій визначає, як Web-сервіси інформують про своє існування і як вони знаходять один одного в мережі. Щоб сервіси могли знайти один одного в мережі, використовується протокол Universal Description, Discovery and Integration (*UDDI*), який визначає реєстр і відповідний протокол для виявлення і доступу до сервісів. *Web Services Inspection Language* є альтернативою UDDI та працює без використання реєстру.

Протоколи безпеки для Web-сервісу починаються зі специфікації WS-Security, яка визначає *маркерну* (*token*) архітектуру для безпечної взаємодії.

На цій основі побудовано шість основних компонентних специфікацій:

− *WS-Policy* і пов'язані з нею специфікації визначають політики правил, за якими взаємодіють Web-сервіси;

− *WS-Trust* визначає довірчу модель для безпечного обміну;

− *WS-Privacy* визначає, як забезпечується конфіденційність інформації;

− *WS-Secure Conversation* визначає як, використовуючи правила, описані в *WS-Policy*, *WS-Trust* та *WS-Privacy* встановити безпечну сесію між сервісами для обміну інформацією;

− *WS-Federation* визначає правила розподіленої ідентифікації та керування нею;

− *WS-Authorization* здійснює авторизацію для доступу й обміну даними.

Після моделі безпеки використовуються прикладні специфікації, включаючи *Business Process Execution Language for Web Services* (*BPEL4WS*), які визначають операції потоків робіт, *WS-Transaction* і *WS-Coordination* та разом обробляють розподілені транзакції.

4.6.5. Використання Web-сервісів у прикладних програмах

При створенні прикладного програмного забезпечення з використанням Web-сервісу застосовують сучасне сімейство комунікаційних протоколів, яке дозволяє прикладним сервісам спілкуватися одному з одним. Такий підхід за останні декілька років набув достатньо значного розвитку у вигляді певних інструментів, які дозволяють розробникам створювати взаємодійні сервіси і складні прикладні програми.

Подібне використання web-сервісів лише як комунікаційного протоколу неефективно, оскільки не використовує сервіс-орієнтованої

архітектури (SOA), що описує в цілому систему сервісів, які динамічно знаходять один одного та спільно виконують певну послідовність дій, об'єднуючись при цьому різними способами. Така модель дозволяє повторно використовувати технології та програмне забезпечення, що зумовлює еволюцію методів створення прикладних програм, їх розгортання і використання.

Використання сервіс-орієнтованої архітектури дозволило застосовувати розподілені обчислення в реальних виробничих процесах, зокрема завдяки розробці прикладного програмного забезпечення у вигляді Web-сервісів.

Для ефективного функціонування Web-сервісу застосовують технології їх розподіленої взаємодії, такі як «сервісна шина підприємства» (*Enterprise Service Bus, ESB*), яка є загальною мережею для взаємодії розподілених сервісів.

Підхід, за якого сервіси розглядають як будівельні блоки, з яких можуть бути зібрані програми, формує найбільш високий рівень абстракції у разі створення програмної системи на базі моделі SOA та множини взаємодіючих компонент у вигляді Web-сервісів, зменшуючи трудовитрати у порівнянні з традиційними методами написання коду програм. Аналізуючи та узгоджуючи інтерфейси взаємодії можна побудувати програму без написання програмного коду, створення якого не є ефективним, оскільки сервіси можуть бути написані різними мовами програмування і для різних платформ. Блоки об'єднують в межах потоку обчислень, який визначає логіку програми, а інші інструментальні засоби використовують для моніторингу ефективності потоку кожного сервісу або групи сервісів. У такому разі розробники все менше використовують звичайні мови програмування, працюючи з архітектурою, керованою моделями (*Model-Driven Architecture, MDA*), що допомагає створювати програмне забезпечення, яке більш точно

відповідає вимогам проекту та функціонує поверх розподіленого програмного середовища проміжного рівня, зокрема ESB.

4.7. Висновки

1. Розподілені системи об'єктів майже завжди грунтуються на моделі віддалених об'єктів, що вимагає додаткового налаштування, наприклад, для підтримки кешування або реплікації.

2. У технології CORBA для зміни вихідних та вхідних запитів використовуються перехоплювачі. У DCOM призначений для користувача маршалінг вимагає налаштування клієнтського замісника. У Globe налаштування виконується за допомогою різних локальних об'єктів, кожен з яких розміщується на власній машині, а всі разом вони утворюють єдиний розподілений об'єкт.

3. Окрім синхронних звертань до методів, розподілені системи об'єктів CORBA або DCOM підтримують альтернативні засоби звертання, включаючи події й асинхронні виклики методів. Globe не надає таких альтернатив і, по суті, підтримує лише синхронні звертання.

4. Організація серверів об'єктів у різних системах майже однакова, зокрема сервер здатний підтримувати більш ніж один об'єкт. Відмінності виявляються у гнучкому налаштуванні сервера: у CORBA гнучкість забезпечується за допомогою адаптерів об'єктів; DCOM надає стандартний сервер об'єктів, який може бути перебудованим під конкретне прикладне програмне забезпечення; в Globe сервери об'єктів порівняно прості, оскільки певні спеціальні властивості реалізують всередині розподілених об'єктів.

5. Відмінності в механізмах іменування розподілених систем об'єктів відбуваються на рівні посилань на об'єкти. Підтримка осмислених імен майже однакова: в CORBA, і в DCOM внутрісистемні

посилання на об'єкти залежать від їх місця розташування і містять інформацію про місце дислокації серверів, на яких перебувають об'єкти. Натомість посилання у Globe не залежать від місця розташування, але для цієї системи потрібна глобальна служба локалізації, яка дозволяє здійснювати посилання до контактних адрес, які визначають, де і як можна знайти вказаний об'єкт.

6. Реплікація в CORBA підтримується лише для задоволення вимоги щодо відмовостійкості. Технологія DCOM взагалі не підтримує реплікацію, розробник прикладного програмного забезпечення має у разі потреби використовувати спеціалізований сервер реплікації або явну програмну реплікацію. У Globe реплікацію підтримує кожний об'єкт окремо за допомогою підоб'єкта реплікації, який реалізує конкретний протокол реплікації об'єкта, частиною якого він є.

7. Питання дотримання вимог відмовостійкості та реплікації тісно пов'язані, але мають певні особливості у разі їх реалізації в різних технологіях. CORBA забезпечує відмовостійкість за допомогою служби реплікації, в основі якої лежить базова служба надійної групової розсилки. У DCOM відмовостійкість підтримується за допомогою транзакцій (автоматичних), а Globe підтримує відмово стійкість лише через реплікацію, а механізмів відновлення не має.

8. Кожна з розподілених систем має засоби захисту, зокрема CORBA пропонує повністю захищену архітектуру, яка дозволяє забезпечити індивідуальний захист кожного об'єкта; у DCOM захист відбувається за рахунок організації доступу до наявних служб захисту, наприклад Kerberos; у Globe також захищається кожний об'єкт окремо, при цьому об'єктам надаються засоби для прив'язки до наявних служб захисту.

9. Web-сервіси є найбільш сучасною технологією побудови системи розподілених об'єктів, можуть бути розроблені з використанням різних мов програмування, включаючи Java, Python, Perl, C#, Basic та інші.

10. Web-сервіси використовують в Internet для побудови Web-орієнтованого програмного забезпечення, яку забезпечує найкращу взаємодію web-сервісів один з одним. Так, насьогодні більшість web-сервісів ґрунтуються на програмному забезпеченні, що працює на серверах програмного забезпечення проміжного рівня, зокрема Websphere, Apache та інших, які не є обя'зковими для застосування, але деякі з розповсюджених інструментальних засобів спроектовані під ці середовища.

4.8. Запитання для самоконтролю

1. Що таке розподілена система? Яку архітектуру вона має?
2. Які переваги використання розподілених систем?
3. Що таке тонкі клієнти?
4. Що таке CORBA? Яку архітектуру вона має?
5. Які складові має система CORBA?
6. Як реалізується сховище інтерфейсів у CORBA?
7. З яких компонентів складається модель DCOM?
8. Як побудовано технологію Globe?
9. Яку структури має розподілений об'єкт Globe?
10. Які переваги та недоліки має Globe?
11. Які переваги та недоліки DCOM?
12. Які переваги та недоліки CORBA?
13. Що таке Web-сервіс?
14. Які стандарти використовуються для Web-сервісу?
15. Як взаємодіють стандарти Web-сервісу?
16. Які можливості дає використання Web-сервісу?
17. Навести приклади протоколів безпеки Web-сервісу.
18. Що таке SOA?
19. Порівняйте технологію Web-сервісів та інші технології.

20. Як використовуються Web-сервіси у прикладних програмах?
21. Як Web-сервіси використовують модель SOA?
22. Що таке контракт сервісу?
23. Як будують розподілені системи з використанням моделі SOA?

5. РОЗПОДІЛЕНІ ФАЙЛОВІ СИСТЕМИ

5.1. Організація даних у розподілених системах

Файлова система (ФС) – спосіб організації даних, який операційна система використовує для збереження інформації у вигляді файлів на носіях інформації, а також сукупність файлів та директорій, які розміщуються на логічному або фізичному пристрої.

Залежно від організації файлів на носіях даних, розрізняють такі ФС:

− ієрархічні – дозволяють розміщувати файли в каталоги, їх широко застосовують сучасні операційні системи (рис.5.1);

− однорівневі – не використовують каталогів, на відміну від ієрархічних ФС;

− кластерні – дозволяють розподіляти файли між кількома однотипними фізичними пристроями;

− мережні – забезпечують механізми доступу до файлів однієї машини з інших машин мережі;

− розподілені – забезпечують зберігання файлів через їх розподіл між кількома машинами мережі.

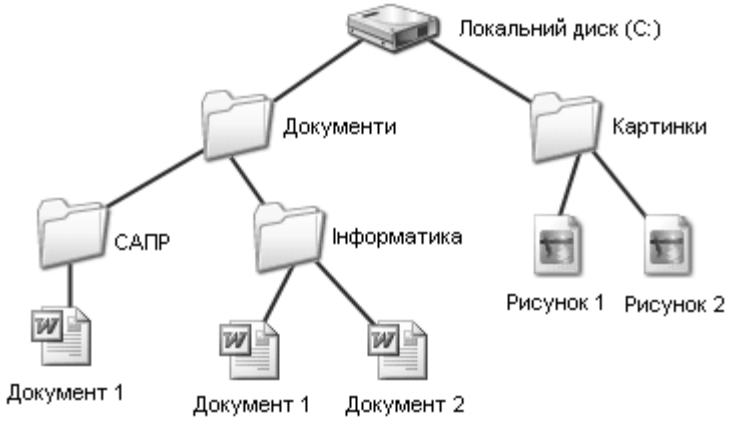


Рис. 5.1. Ієрархічна файлова система

5.1.1. Системи зберігання даних та файлові системи

Системи зберігання даних (*Storage Systems*) (наприклад, окремі дискові пристрої – жорсткі диски) користувачі та програми нині використовують як виділені пристрої зберігання даних. Кожний дисковий пристрій має деяку кількість блоків – фіксований розмір сегментів даних, наприклад 1К (1024 байти). Коли дисковий пристрій під'єднано до комп'ютера, він може обробляти лише найпростіші запити, такі як:

- *READBLOCK (12345)* – прочитати блок № 12345 і відправити блок даних у комп'ютер;
- *WRITEBLOCK (765645)* – отримати дані з комп'ютера і зберегти їх у блок № 765645.

Диски можуть бути підключені до комп'ютерів за допомогою Integrated Drive Electronics (*IDE*), Small Computer Systems Interface (*SCSI*) або Fiber Distributed Data Interface (*FDDI*) інтерфейсів, які використовуються для передачі команд і даних на диски, а також для завантаження даних і кодів команд завершення з дисків. Системи зберігання даних не створюють ніяких інших елементів, а це означає, що дисковий пристрій не може створювати «файли» або «файлові директорії», тому що єдині об’єкти, з якими такі пристрої працюють, – це блоки, а всі функції, що вони виконують, – це читання і запис цих блоків (кластерів).

5.1.2. Файлова система окремої операційної системи

Будь-яка сучасна операційна система (ОС) має компонент – файлову систему. Цей компонент є частиною ядра операційної системи та реалізує такі поняття, як «файли» і «файлові каталоги». Є багато

різних файлових систем, які використовують різні методи і алгоритми, але основні функції виконує більшість файлових систем:

− має таблицю File Allocation Table (*FAT*), що містить інформацію, яка пов'язує логічні файли зі збереженням номера блока на фізичному диску. Наприклад, FAT може вказати, що «*file1*» зберігається на п'яти блоках диска з номерами 123400, 123405, 123401, 177777, 123456 і «*file2*» файл зберігається на шести блоках диска з номерами 323400, 323405, 323401, 377777, 323456, 893456;

− керує списком усіх невикористовуваних блоків на диску й автоматично розподіляє нові блоки, коли файл збільшується в розмірах, і повертає блоки до списку невикористовуваних блоків, коли файл зменшується в розмірах, або коли видаляється;

− керує процесами читання/записування файлів на диск, перетворює запити читання/записування в одну або кілька операцій читання і запису, використовуючи інформацію з таблиці розміщення файлів;

− керує файловими директоріями, зберігає в них інші файли;

− підтримує кешування, тобто коли нова інформація записується у файл, зберігає його на диску, а також його копію в буфері пам’яті – кеші. Коли файл зчитує з диску прикладна програма, то він також копіюється в «кеш-буфер». Коли програмі необхідно прочитати частину кешованих файлів, ФС просто витягує цю інформацію зі свого кешу буферів, а не повторно відкриває його з диску, що значно прискорює роботу. На рис. 5.2 показано принцип роботи ФС на окремій ОС, з якого зрозуміло, що у разі необхідності прочитати дані з диска прикладна програма звертається до ФС, а ФС за допомогою FAT таблиці або кешу зчитує файл.

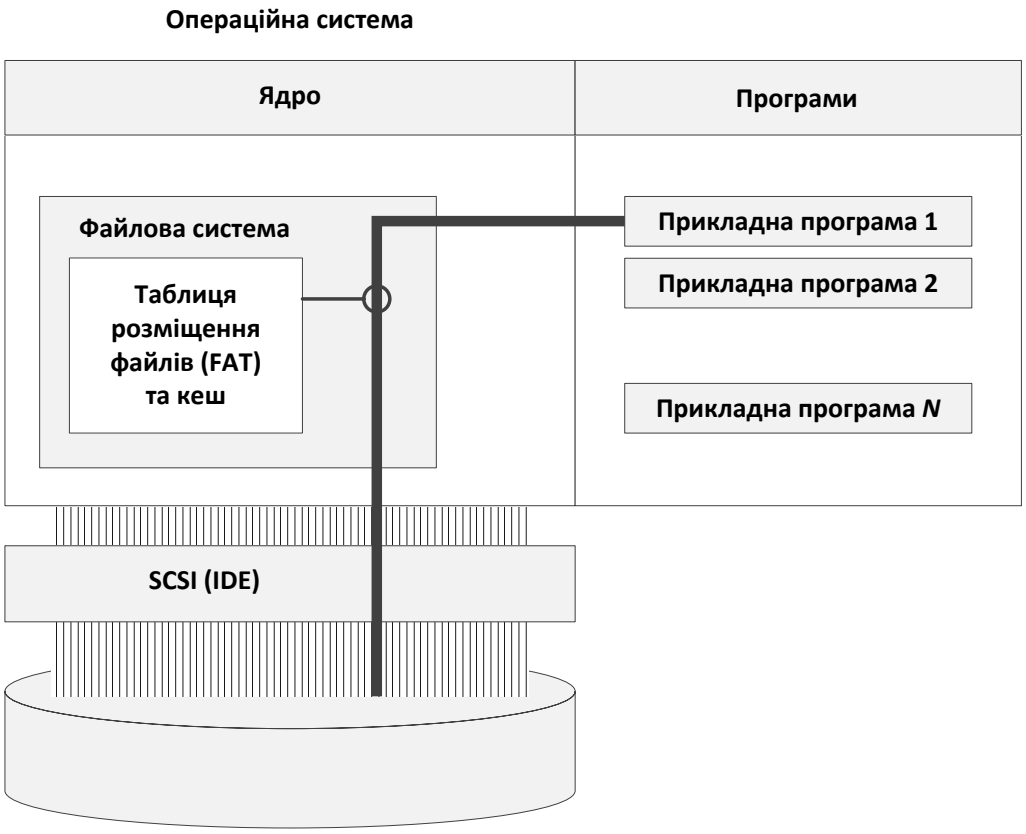


Рис. 5.2. Принцип роботи файлової системи на окремій операційній системі

5.1.3. Мережні файлові системи

Коли комп'ютери або сервери потребують використання деяких однакових даних, то можуть бути використані мережні ФС, зокрема *Network File System* (*NFS*) або *Network Attached Storage* (*NAS*).

Мережна ФС будується на основі файлового сервера та мережі. **Файловий сервер** – звичайний комп'ютер, або комп'ютер зі спеціалізованою ОС, що має дискові масиви під керуванням ФС.

Мережна ФС працює всередині ядра ОС клієнтської машини мережі, що перенапрямляє запити на файловий сервер через мережу, як показано на рис. 5.3.

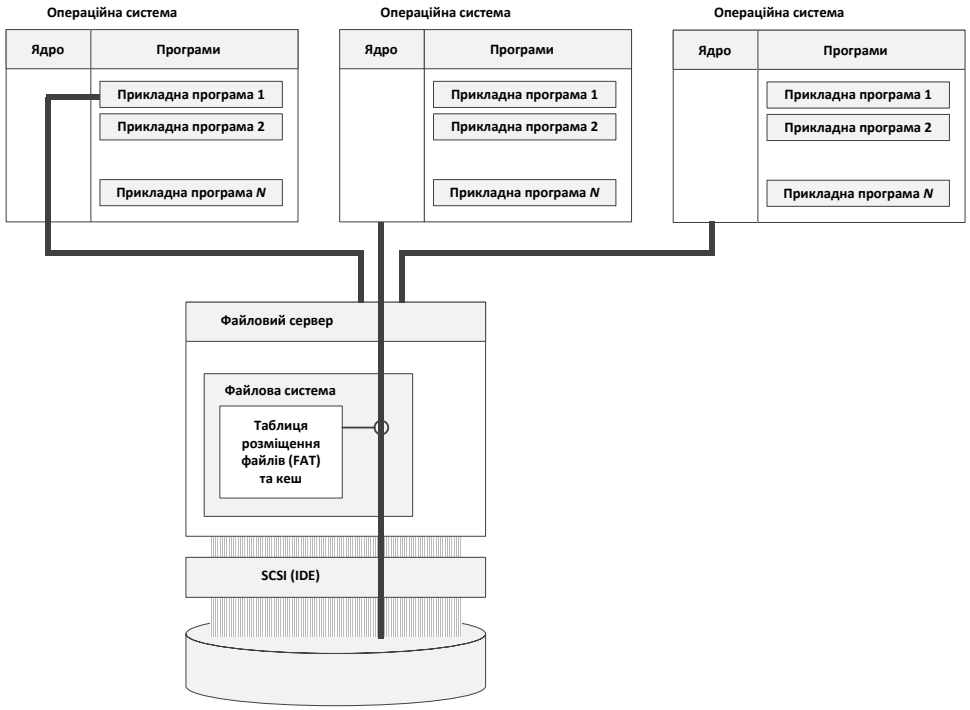


Рис. 5.3. Принцип роботи мережної файлової системи

5.1.4. Кластерні файлові системи

Кластерні ФС – це програмні продукти, що дозволяють створювати мультикомп'ютерні системи з «розшареними» (*shared*) дисками та зазвичай реалізуються як «модернізація» деяких регулярних ФС. Кластерні ФС використовують частину мережі, що об'єднує сервери, і синхронізують їх діяльність за принципом роботи, показаним на рис. 5.4, коли кожний сервер може рівноправно звернутись до усіх наборів даних.

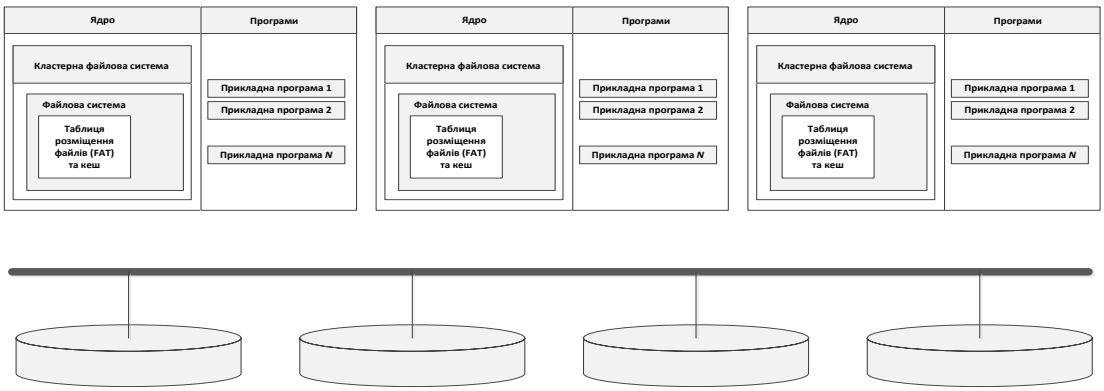


Рис. 5.4. Принцип роботи кластерної файлової системи

В таблиці 5.1 наведено приклади кластер них ФС у поєднанні з ОС, що їх використовують.

Таблиця 5.1. Приклади кластерних файлових систем та операційні системи, на яких вони реалізуються

Кластерна ФС	**Операційна система**
Tru64 Cluster 5.x	HP Tru64
VERITAS ClusterFileSystem	SunSolaris, HP/UX
SunCluster 3.0	SunSolaris
GeneralizedParallelFileSystem (GPFS)	IBM AIX, Linux
DataPlow	Linux, Solaris, Windows, IRIX
PolyServe	Linux
GFS	Linux
NonStopCluster	Unixware

5.1.5. Розподілені файлові системи

Ключовим компонентом будь-якої розподіленої системи є ФС. Як і в централізованих системах, у розподіленій системі функцією ФС є зберігання програм і даних та надання доступу до них у міру необхідності. ФС підтримує одна або більше машин, які називають файл-серверами, що перехоплюють запити на читання або запис файлів, які надходять від клієнських машин. Кожен запит перевіряється і виконується; надсилається відповідь. Файл-сервери зазвичай містять ієрархічні ФС, кожна з яких має кореневий каталог і каталоги більш низьких рівнів. Робоча станція може приєднувати і монтувати ці ФС до своїх локальних файлових систем, при цьому монтовані ФС залишаються на серверах.

Часто мережні ФС називають *розподіленими файловими системами (РФС)*. Цей термін відображає той факт, що велика кількість файлових систем мають набагато більше можливостей, ніж

просте передавання даних мережею. Носії даних, пов'язані з цими файловими системами, не обов'язково розташовані на одному комп'ютері – вони можуть бути розподіленими між багатьма комп'ютерами. Наприклад, розподілені ФС OpenAFS і Coda мають власні механізми керування розділами, які спрощують можливості зберігання загальнодоступної інформації. Вони також підтримують *дублювання* – здатність робити копії розділів і зберігати їх на інших файлових серверах. Якщо один файловий сервер стає недоступним, то до даних, що зберігаються в його розділах, можна отримати доступ за допомогою наявних резервних копій цих розділів.

Сумісне використання даних – фундаментальна вимога розподілених систем. Розподілені ФС дозволяють декільком процесам протягом тривалого часу спільно працювати зі спільними даними, забезпечуючи їх надійність і захищеність. З цієї причини вони нерідко використовуються як базовий рівень розподілених систем і прикладних програм.

Приклад. Найвищий рівень організації у файловій системі – це розподіл дискового простору між клієнтами (як у файлових системах Windows), робочі станції клієнтів для отримання доступу до цих даних повинні обов'язково під'єднатися до свого розділу та позначити його окремою літерою у своїй локальній розкладці (зокрема, диск E , F, G, і та ін.). Букви мають бути співставленими з мережними розділами, призначеними для користувача і груповими профілями Windows (для стандартизації). Але, на жаль, не на всіх комп'ютерах розміщення літер може бути однаковим, зокрема на комп'ютері з великою кількістю жорстких дисків і розділів потрібні букви можуть бути зайняті, тому доведеться давати мережним розділам інші позначення.

Навпаки, Unix – це ієрархічна файлова система, до якої додаткові розділи додаються за допомогою монтування їх до наявної директорії, що дозволяє ефективно долучити будь-яке джерело даних у будь-яку наявну файлову систему. Якщо монтують нове джерело інформації до каталогу, який є частиною розподіленої файлової системи, він відразу ж стає доступним усім клієнтам цієї розподіленої системи.

Сучасні розподілені файлові системи типів OpenAFS або Coda охоплюють спеціальні сервіси для керування розділами, що дає змогу змонтувати розділи різних файлових серверів у центральну ієрархію директорій, підтримувану файловими системами. OpenAFS використовує центральний каталог */afs*, а Coda - */coda*. Ці ієрархії директорій доступні всім клієнтам розподіленої файлової системи і виглядають однаково на будь-якій клієнтській робочій станції, що дає можливість користувачам працювати зі своїми файлами однаково на будь-якому комп'ютері, оскільки всі файли користувачів зберігаються на сервері.

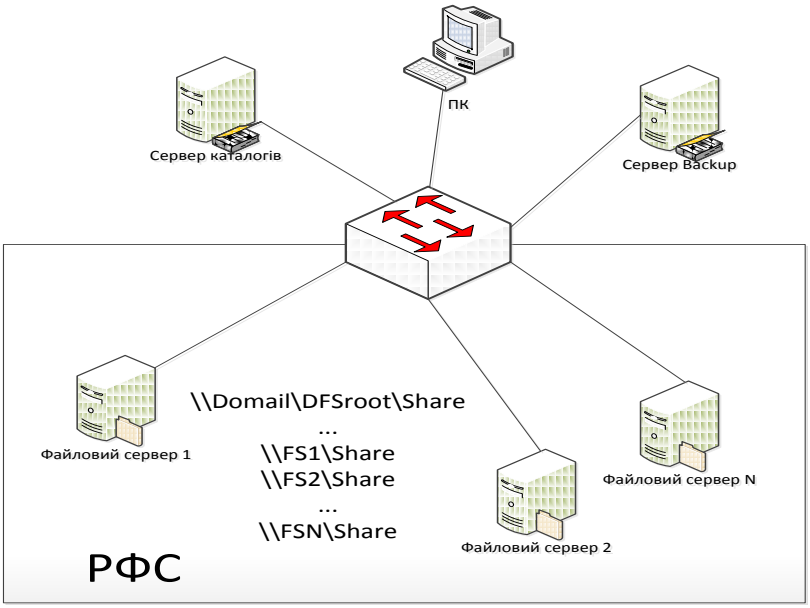


Рис. 5.5. Приклад розподіленої файлової системи

Розподілені файлові системи, приклад організації яких подано на рис. 5.5, є особливо зручними у процесі координації роботи між групами, розташованими в різних містах, державах або навіть у різних країнах. Перевага полягає в тому, що загальні дані завжди доступні мережею незалежно від їх місцезнаходження, при цьому дані, перебуваючи на різних серверах, утворюють враження, що знаходяться в одному місці.

Розглянемо поширені розподілені ФС – мережну файлову систему *Network File System* (*NFS*) компанії Sun Microsystem та Coda, які мають суттєві відмінності, а також Plan 9, xFS, SFS та Ceph.

Характерною особливістю та перевагою системи NFS є її здатність ефективно функціонувати в мережі.

Coda – це нащадок мережної файлової системи *Andrew File System* (*AFS*), великомасштабної системи, яку розробляли виключно для того, щоб забезпечити максимальну масштабованість. Від множини інших файлових систем Coda відрізняється підтримкою безперервних операцій, які функціонують, не обмежуючись розмірами сегментів мережі, що дуже зручно для мобільних користувачів, які вимушені часто від'єднуватися від мережі.

Plan 9 – це розподілена система, в якій усі ресурси розглядаються як файли, тобто її можна вважати розподіленою системою файлів.

XFS – характеризується тим, що в ній немає серверів, а файлову систему реалізують клієнти.

SFS – вирізняється серед інших розподілених файлових систем маштабованою системою захисту.

Система Ceph – може використовуватися як у системах, які складаються з декількох вузлів, так і в системах, що містять кілька тисяч вузлів, тобто загальний обсяг сховища даних може вимірюватися петабайтами. Система містить вбудовані механізми реплікації даних, які забезпечують її надзвичайно високу живучість, оскільки у разі додавання або видалення нових вузлів масив даних автоматично ребалансується з урахуванням нововведень.

5.2. Базові принципи побудови розподілених файлових систем

Побудова розподіленої ФС здійснюється за такими принципами як: децентралізація, реструктуризація, раціональне використання дискового простору, реплікація даних і локальне кешування, ланцюги блоків, контроль версії блока, можливість апаратного прискорення (рис. 5.6).

Рис. 5.6. Базові принципи побудови розподілених файлових систем

Децентралізація є одним з базових принципів побудови розподілених ФС через те, що розташування даних в одну місце обов'язково зумовить збільшення навантаження через підвищення кількості запитів на доступ до даних. Наприклад, на кластері використовується файлова система *NFS*, яка дозволяє змонтувати домашню папку *(/home*) на всі обчислювальні вузли й тим самим надати доступ до загальних дискових ресурсів. Недоліком такої організації системи є централізація зберігання даних, яка зумовить необхідність звертань усіх користувачів до одного дискового простору, створюючи «вузьке місце», що обмежує пропускну здатність і продуктивність кластера (рис.5.7).

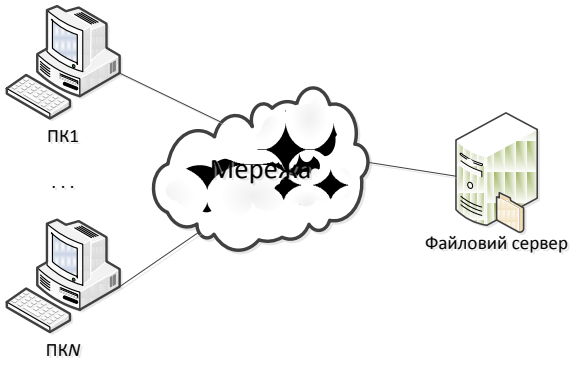


Рис. 5.7. Приклад централізації

У разі інтенсивної обробки дані будуть пересилатися між файловим сервером і обчислювальними вузлами. Продуктивність буде обмежена пропускною здатністю одного мережного інтерфейсу файлового сервера або швидкістю лінійного записування на диск. Таке обмеження можна усунути за рахунок децентралізації зберігання. Всі

обчислювальні вузли кластера під'єднані через комутатор, який забезпечує одночасне передавання даних між непересічними маршрутами. У такий спосіб у системі з реплікацією (клонуванням) даних на кілька файлових серверів можна підвищити продуктивність системи. Приклад організації розподіленої файлової ситеми подано на рис. 5.8.

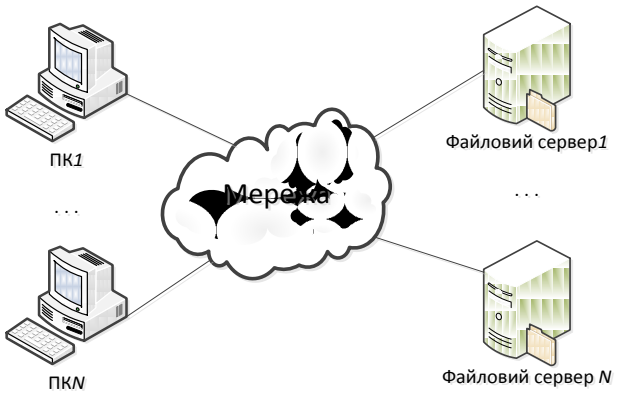


Рис. 5.8. Децентралізація в розподіленій файловій системі

Підвищення продуктивності в розподіленій ФС є її перевагою, але при цьому виникає і недолік, яким є необхідність реалізації розподіленого зберігання, яке вимагає забезпечення одночасного доступу на зміну даних, для чого застосовують блокування доступу на час модифікації даних.

Реструктуризація. Топологія мережі й кількість вузлів у мережі можуть змінюватися у процесі роботи, не зупиняючи функціонування системи, тобто не має бути одного окремого вузла, який відповідає за кореневу папку.

Раціональне використання дискового простору. Розподілене зберігання даних, коли дані рівномірно розподілені вузлами та не мають копії, знижує надійність системи, оскільки неробочий стан одного вузла спричиняє неробочий стан системи в цілом, і відповідно зниження швидкодії одного з вузлів - зниження швидкодії системи в цілому. Звідси виникає потреба клонувати дані (виконувати реплікацію), щоб дані були доступними завжди одночасно з декількох вузлів.

Реплікація даних и локальне кешування. Кожний вузол повинен ухвалювати рішення щодо того, чи є потреба у копіюванні (реплікації) даних. Кожна операція копіювання має задовольняти одній з поставлених цілей і підвищувати впорядкованість системи. Цінність і пріоритет операції обчислюють за умови, що кожній цілі відповідає математичний критерій необхідності виконання операції.

Цілі реплікації такі: відмово стійкість, швидкодія, оптимізація топології розміщення даних. Організацію мережі з реплікацією даних показано на рис. 5.9, де один файловий сервер є активною копією даних, з якою взаємодіють робочі станції, а другий – пасивною.

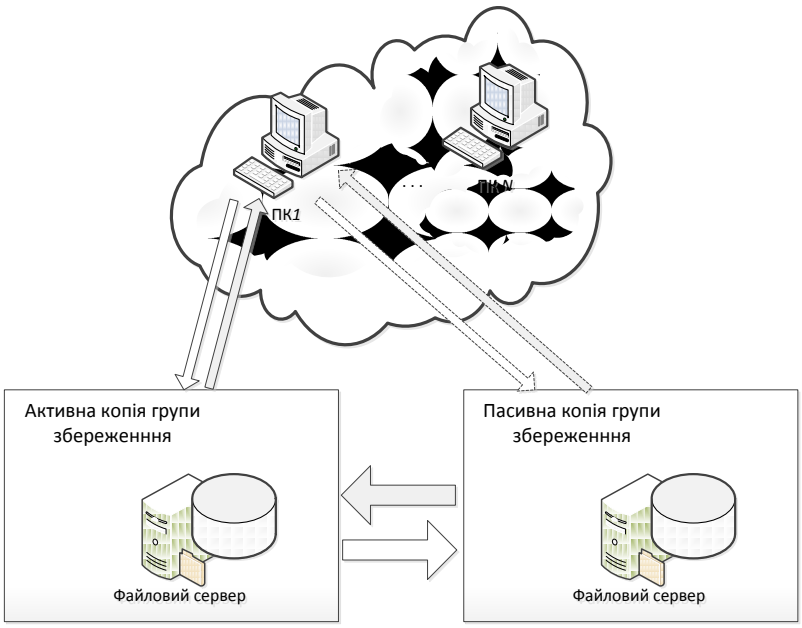


Рис. 5.9. Реплікація даних

Кешування – це зберігання часто використовуваної інформації в такому місці, звідки її у разі потреби можна швидко і легко отримати. Локальне кешування означає використання ресурсів локального комп’ютера (жорсткий диск, оперативна пам’ять) для тимчасового та швидкого доступу до даних.

Ланцюги блоків. Реалізація файлової системи грунтується на концепції блоків, наприклад, блоки використовують для запису, читання та зберігання даних на диску або флеш пам'яті, коли запис виконується блоками та секторами фіксованої довжини.

Під час виконання обчислювальних завдань можуть генеруватися файли дуже великих розмірів, реплікація яких неможлива, оскільки зміни й доповнення у вихідні дані вносяться швидше, ніж здійснюється реплікація.

Зберігання ланцюгів блоків, а не файлів, які мають різні розміри, дозволяє надавати доступ до фрагментів файлів ще до завершення операції копіювання.

Контроль версії блока. Для підвищення продуктивності системи застосовують контроль версії блока. Якщо версію блока не змінено, то немає потреби копіювати блок. За рахунок зменшення операцій копіювання блоків, які не змінювалися, вдається уникнути повторного копіювання даних, що розвантажує мережний трафік.

Ідею організації контролю версії блока взято з протоколу *http*, яка полягає у тому, що власник блока повідомляє під час пересилання деяке число, яке характеризує версію блока. У разі повторного запиту того самого блока, власникові відсилається версія локальної копії, у відповіді на запит отримується або вміст блока, або повідомлення, що дані залишилися без змін. У такому механізмі контролю версії немає прив'язки до необхідності внесення змін одночасно у всіх вузлах і потреби у контролі версії на локальному вузлі.

Розробляючи концепцію розподіленої файлової системи, значну увагу при приділяють можливості апаратної реалізації протоколів обміну даними.

5.3. Мережна файлова система компанії Sun Microsystems

Систему NFS спочатку було розроблено компанією *Sun Microsystems* для використання в робочих станціях на основі *UNIX*, але потім реалізовано і на багатьох інших платформах. Основна концепція, яка застосовується у NFS, полягає в тому, що кожен файловий сервер

має стандартне подання власної локальної файлової системи. Інакше кажучи, не важливо, як саме реалізована локальна ФС, оскільки кожен сервер NFS підтримує одну модель, яка містить протокол зв'язку, який дозволяє клієнтам мати доступ до файлів, що зберігаються на сервері. Такий підхід дає змогу різнорідним наборам процесів, які, можливо, працюють під керуванням різних операційних систем і на різних машинах, спільно використовувати єдину файлову систему.

Network File System (NFS) – це не тільки файлова система, а й набір протоколів, які створюють перед клієнтом уявлення розподіленої файлової системи. У цьому сенсі NFS можна порівняти із системою CORBA, яка також існує тільки у вигляді описів. Як і CORBA, NFS має дуже багато реалізацій, що працюють під керуванням різних операційних систем на різних машинах. Протоколи NFS розроблялися так, щоб їх різноманітні реалізації з легкістю могли працювати спільно.

Модель, що лежить в основі NFS, – це *віддалена файлова служба* (*remote file service*), у якій клієнти здійснюють прозорий доступ до файлової системи, яку підтримує віддалений сервер, проте зазвичай не обізнані про дійсне місцеположення файлів. Замість цього їм надається інтерфейс файлової системи, схожий на інтерфейс стандартної локальної файлової системи. Зазвичай в інтерфейсі, який є у клієнта, визначені різноманітні операції з файлами, але за реалізацію цих операцій відповідає сервер, тому таку модель називають також *моделлю віддаленого доступу* (*remote access model*), яку показано на рис. 5.10, *а*.

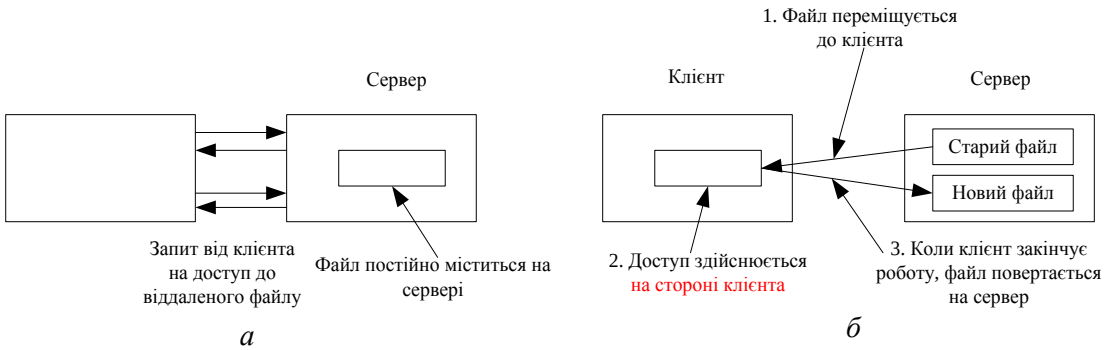


Рис. 5.10. Модель віддаленого доступу (*а*); модель завантаження - вивантаження (*б*)

Натомість у *моделі завантаження-вивантаження* (*upload/download model*) після завантаження файлу із сервера клієнт отримує локальний доступ до нього, як показано на рис. 5.10, *б*. Коли клієнт закінчує працювати із файлом, файл вивантажується назад на сервер, після чого з ним може працювати інший клієнт. Таким чином використовується Internet-служба FTP – клієнт завантажує файл цілком, вносить до нього зміни, а потім повертає на місце.

Система NFS реалізована для багатьох платформ, хоча версії для UNIX переважають. Фактично для всіх сучасних UNIX-систем NFS реалізується відповідно до багаторівневої архітектури, показаної на рис. 5.11.

Клієнт, дістаючи доступ до файлової системи, використовує системні виклики локальної операційної системи. Проте інтерфейс локальної файлової системи UNIX замінюється інтерфейсом *віртуальної файлової системи* (*Virtual File System, VFS*), який у певний час стає де-факто стандартом інтерфейсу розподілених файлових систем. Операції з інтерфейсом VFS передаються локальній файловій системі або спеціальному виокремленому компоненту, який відомий під назвою *клієнт NFS* (*NFS client*) і відповідає за надання доступу до файлів, що зберігаються на віддаленому сервері. У NFS уся взаємодія між клієнтом і сервером здійснюється за допомогою викликів віддалених процедур RPC. Клієнт NFS реалізує операції файлової системи NFS у формі запитів RPC, які надсилаються серверу.

Відзначимо, що набір операцій, який надає інтерфейс VFS, може відрізнятися від набору, який надає клієнт NFS. Основна перевага VFS полягає в тому, щоб приховати відмінності між різними файловими системами.

Подібним чином організований і сервер. Сервер NFS відповідає за обробку запитів клієнтів. Заглушка витягує запити RPC із повідомлень, а сервер NFS перетворює їх на коректні операції з файлами інтерфейсу VFS, які передаються на рівень VFS, що відповідає за реалізацію локальної файлової системи, яка забезпечує роботу з реальними файлами.

Важлива позитивна ознака цієї схеми полягає в тому, що NFS фактично не залежить від локальних файлових систем. Не важливо, яку файлову систему – UNIX, Windows 2000 або навіть архаїчну MS-DOS – реалізує операційна система клієнта або сервера. Єдине, що важливе, – щоб ці ФС не суперечили моделі файлової системи, яку підтримує NFS. Так система MS-DOS з її короткими іменами файлів не може підтримувати повну прозорість реалізації сервера NFS.

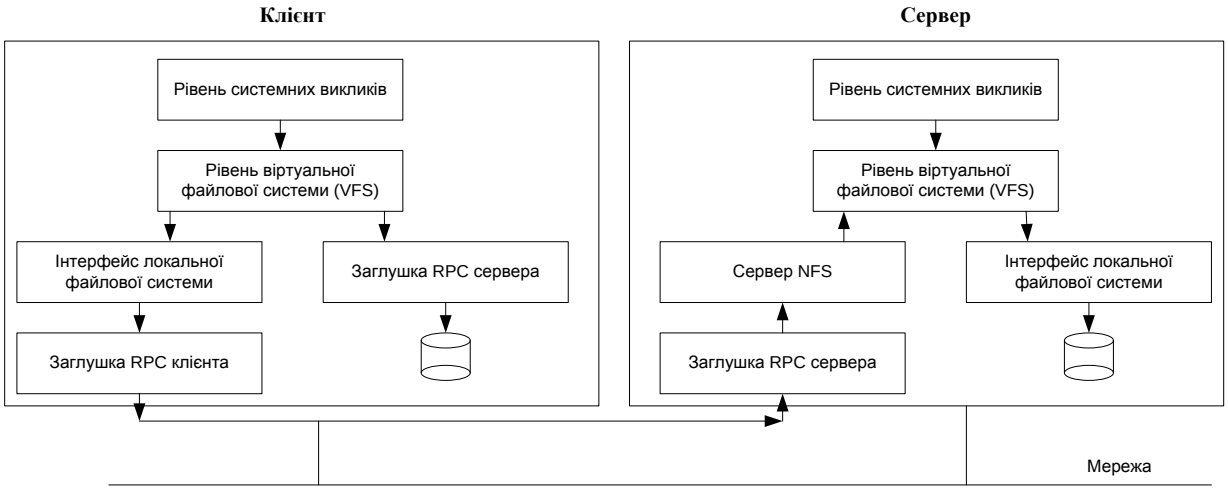


Рис. 5.11. Базова архітектура NFS для UNIX-систем

Модель файлової системи, яку підтримує NFS, аналогічна моделі UNIX-систем, де файли трактуються як послідовності байтів, які не інтерпретуються, та організовані у вигляді ієрархічного графа імен, вузлами якого є каталоги і файли. Файлова система NFS підтримує як

задані явно посилання, так і символічні посилання, які використано у стандартних файлових системах UNIX.

Файли мають імена, але доступ до них здійснюється за допомогою UNIX-подібних *дескрипторів файлів* (*file handle*), тобто клієнт має за допомогою служби іменування отримати асоційований з ним дескриптор файлу, а кожний файл використовує декілька атрибутів, значення яких можна зчитувати і змінювати.

Операція *create* використовується для створення файлів, але має різну функціональність у версіях файлових систем, наприклад версії 3 і 4 NFS. Так, у версії 3 ця операція створює звичайний файл, тоді як спеціальні файли створюються за допомогою іншої операції; операція *link* дозволяє створювати задані явно посилання, *symlink* - символічні посилання; *mkdir* - вкладені каталоги. Спеціальні файли, зокрема файли пристроїв, сокети й іменовані канали, створює операція *mknod*.

Натомість, версія 4 NFS застосовує операцію *create* для створення нестандартних файлів, зокрема символічних посилань, каталогів і спеціальних файлів. Задані явно посилання, як і за версією 3, створює операція *link*, а звичайні файли - операція *open*, яка є новою для NFS і радикально змінює методи роботи із файлами, характерні для попередніх версій. До версії 4 система NFS створювалась так, що її файлові сервери не зберігали інформацію про стан, у результаті чого працюючи з одним і тим самим файлом, сервери зберігають стан файлу між операціями. Операцію *rename* використовують для зміни імені наявного файлу. Файли видаляються за допомогою операції *remove*, яку у версії 4 використано для видалення файлів будь-яких типів. У попередній версії видаляла вкладені каталоги окрема операція *rmdir*. Файли видаляються за іменем, у результаті чого кількість заданих явно посилань на них зменшується на одиницю. Якщо кількість посилань дорівнює нулю, то файл можна знищити.

Версія 4 NFS дозволяє клієнтам відкривати і закривати звичайні файли, у такому разі відкриття неіснуючого файлу зумовлює його створення. Для відкриття файлу клієнт указує його ім'я, а також необхідні значення атрибутів. Наприклад, клієнт може вказати, що файл варто відкрити для читання. Якщо файл успішно відкрито, то клієнт може працювати із цим файлом за допомогою дескриптора файлу, який використовується також для закриття файлу (операція *close*). За допомогою операції *close* клієнт повідомляє сервер, що доступ до файлу йому більше не потрібний, а сервер, у свою чергу, може скинути інформацію про стан файлу, яку він підтримував для організації доступу клієнта до цього файлу.

Операція *lookup* використовується для отримання дескриптора файлу за заданим повним шляхом до нього. У NFS версії 3 операція *lookup* не визначала імена за монтажною точкою (каталогом, який є посиланням на вкладений каталог у чужому просторі імен). Припустімо, ім'я */remote/vu* відповідає монтажній точці у графі імен. У разі дозволу імені */remote/vu/mbox* операція *lookup* у NFS версії 3 повертає дескриптор файлу монтажної точки */remote/vu* разом із залишком повного імені (тобто *mbox*). Після цього клієнт повинен був явно вказати файлову систему, необхідну для завершення пошуку імені, тобто ФС у такому розумінні – це набір файлів, атрибутів, каталогів і блоків даних, які спільно реалізовані у вигляді логічного пристрою з блоковою структурою даних.

У версії 4 операція *lookup* визначає ім'я цілком навіть у разі переходу через монтажні точки. Відзначимо, що це можливо лише за умови, що чужа ФС уже змонтована у відповідну монтажну точку. Клієнт спроможний виявити перехід через монтажну точку, відстежуючи ідентифікатор файлової системи, який буде повернений йому після закінчення дозволу імені.

Для читання елементів каталогу використовують окрему операцію *readdir*, яка повертає список пар (ім'я, дескриптор файлу) разом зі значеннями атрибутів, запитаних клієнтом. Клієнт може також визначити, скільки елементів йому слід повернути. Операція повертає зсув, який може бути використаний під час наступного виклику *readdir* для читання нової послідовності елементів.

Операція *readlink* використовується для читання даних, що містяться в символічному посиланні та відповідають повному імені, яке потім можна визначити звичайним способом. Відзначимо, що операція пошуку не може працювати із символічним посиланням, у разі виявлення якого, дозвіл імені припиняється, і клієнт повинен спочатку викликати *readlink*, щоб отримати ім'я, з якого він зможе продовжити процедуру дозволу імен.

Способи роботи з атрибутами файлів визначають ще одну серйозну відмінність між NFS версій 3 і 4, оскільки файли мають різні атрибути. До стандартних атрибутів належать тип файлу (вказує, що файл є каталогом, символічним посиланням, спеціальним файлом тощо); довжина файлу; ідентифікатор файлової системи, в якій міститься файл; час останньої модифікації файлу. Атрибути файлу можна прочитувати і записувати за допомогою операцій *getattr* і *setattr* відповідно.

Читання даних здійснюється за допомогою операції *read*, коли клієнт визначає зсув і кількість байтів, які він хоче прочитати, у відповідь йому ФС повертає реальну кількість прочитаних байтів і додаткову інформація про стан (наприклад, про досягнення кінця файлу).

Запис даних у файл виконується за допомогою операції *write*, коли клієнт знову визначає позицію у файлі, з якою має початися запис, кількість записуваних байтів і дані. Крім цього, він може дати команду серверу забезпечити гарантований запис усіх даних у стійке сховище.

Сервери NFS мають підтримувати пристрої зберігання даних, здатні витримати зникнення живлення, а також відмови операційної системи й апаратного забезпечення.

5.4. Файлова система CODA

Система CODA було розроблено в університеті Карнегі Мелона (Carnegie Mellon University) у 90-х роках XX ст. і нині інтегровано в декілька розповсюджених операційних систем на основі UNIX, наприклад у Linux. CODA суттєво відрізняється від NFS, зокрема високою доступністю. Вимога забезпечити високу доступність змусила розробників застосовувати вдосконалені схеми кешування, які дозволяють клієнтові продовжувати операцію навіть у разі від'єднання від сервера.

CODA розробляли як масштабовану захищену розподілену ФС високої доступності, основне завдання якої полягало в забезпеченні прозорості іменування і локалізації, щоб система здавалася користувачеві подібною до локальної файлової системи. Крім високої доступності, розробники CODA намагалися також забезпечити високий ступінь прозорості відмов системи.

CODA було створено з другої версії файлової системи Andrew (Andrew File System, AFS), запозичивши у неї дуже багато архітектурних рішень. Систему AFS, власне, і створювали для підтримки інфраструктури університету, яка охоплює близько 10000 робочих станцій, яким потрібно забезпечити доступ до системи. Щоб задовольнити ці вимоги, вузли AFS було поділено на дві групи: перша група складається з порівняно невеликої кількості файлових серверів *Vice* із централізованим адмініструванням; друга – це значно більша кількість робочих станцій *Virtue*, які надають користувачам і процесам доступ до файлової системи, як показано на рис. 5.12.

Усі робочі станції *Virtue* виконують прикладний процес *Venus*, роль якого аналогічна ролі клієнта NFS. Функцією цього процесу є надання доступу до файлів, які містяться на файлових серверах *Vice*. *Venus* у CODA відповідає також за те, щоб клієнти мали можливість продовжувати операції у разі, коли доступ до файлових серверів неможливий (тимчасово). Ця додаткова роль значно відрізняє методи, використовувані в CODA, від прийнятих в NFS.

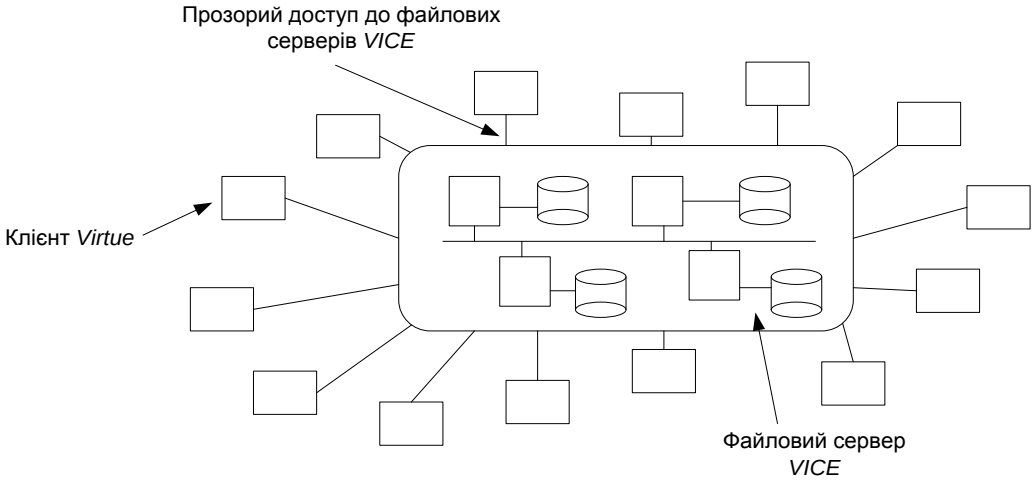


Рис. 5.12. Загальна організація AFS

Внутрішню архітектуру робочої станції *Virtue* показано на рис. 5.13, для якої найбільш важливим є те, що *Venus* - прикладний процес. На системному рівні клієнта введено додатково окремий рівень віртуальної файлової системи (VFS), який перехоплює всі виклики з клієнтських прикладних програм і, як показано на рис. 5.13, передає виклики в локальну файлову систему або процесу *Venus*. Ця структура, включаючи VFS, аналогічна відповідній структурі NFS. *Venus*, у свою чергу, спілкується із файловими серверами *Vice*, використовуючи систему RPC рівня користувача. Система RPC побудована поверх дейтаграмм UDP і підтримує семантику «максимум одного разу».

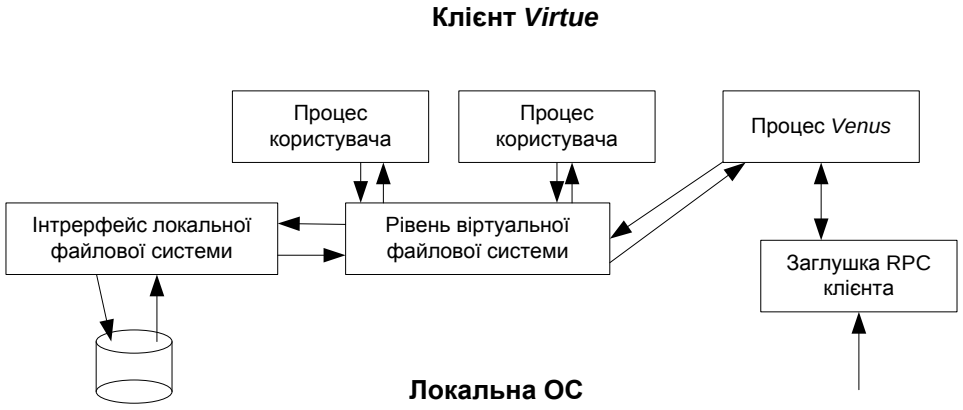


Рис. 5.13. Внутрішня структура робочої станції *Virtue*

На стороні сервера відбуваються три різні процеси: підтримка локального набору файлів, яка виконується власне файловими серверами *Vice*; підтримка аутентифікації, яку здійснюють довірені машини *Vice*; процеси оновлення, потрібні для збереження несуперечливості метаінформації на кожному з серверів *Vice*. Як і *Venus*, файловий сервер працює на призначеному рівні користувача.

CODA надає своїм користувачам традиційну UNIX-подібну файлову систему. На відміну від NFS, CODA має глобальний простір поділюваних імен, який підтримують сервери *Vice*. Клієнти отримують доступ до цього простору імен через спеціальний вкладений каталог у їх локальному адресному просторі, наприклад */afs*. Коли клієнт шукає в цьому вкладеному каталозі ім'я, *Venus* гарантує, що відповідна частина загального простору імен вмонтовується локально.

Загалом розподілена ФС CODA характеризується такими якісними показниками:

- адаптацією смуги пропускання для клієнтів, які мігрують;
- стійкістю до збоїв за рахунок застосування серверів реплікації для читання/запису, розв’язання конфліктів сервер/сервер, визначення місця збою системи, сервера;
- продуктивністю та масштабованістю завдяки кешуванню файлів, директорій та атрибутів на клієнтській стороні для збільшення продуктивності, кешуванню з можливістю відкату виконаного запису;

− захистом, який забезпечує аутентифікація *kerberos*, список контролю доступу (*access control lists, ACL's*);

− добре визначеною семантикою обміну;

− вільно поширюваним вихідним кодом.

Згодом на основі розподіленої ФС Coda було побудовано ще одну розподілену ФС – InterMezzo.

5.5. Plan9 – ресурси як файли

Файлову систему Plan9 було розроблено для застосування декількох централізованих серверів і великої кількості клієнтських машин, як альтернативу мережним операційним системам. Проте замість мейнфреймів або міні-комп'ютерів як сервери передбачено задіювати порівняно потужні й дешеві мікрокомп'ютери. У такому разі сервери також керуються централізовано, а клієнтські машини мають бути простими та орієнтованими на виконання мінімального набору завдань.

Систему розробляла та сама група людей в Bell Labs, яка розробляла UNIX. Велика кількість сучасних локальних розподілених систем побудована за принципом поєднання багаточисельних простих клієнтів і декількох потужних серверів.

Plan9 – не просто розподілена файлова система, а фактично система розподілу файлів. Доступ до всіх ресурсів системи (зокрема, процесів і мережних інтерфейсів) здійснюється однаково, з використанням характерних для файлів синтаксису та операцій. Цей підхід запозичено в системі UNIX, яка також надає ресурсам інтерфейс доступу до файлів, але в Plan9 він значно розширений і витримується більш послідовно. Кожен сервер має ієрархічно організований простір імен для ресурсів, які він контролює. Клієнт може вмонтовувати простір імен сервера локально, вибудувавши таким чином власний

простір імен, за аналогією до підходів, використовуваних у NFS, у якому для забезпечення можливості розподілу імен частину простору імен стандартизовано.

Подібний підхід до побудови файлової системи зумовлює архітектуру, показану на рис. 5.14, де Plan9 складається з набору серверів (*nameserver*, NS), які надають клієнтам ресурси у вигляді локальних просторів імен. Для доступу до ресурсів сервера клієнт вмонтовує простір імен сервера у власний простір імен. Слід зазначити, що різниця між клієнтами і серверами в Plan9 не завжди відчутна, тому що сервери часто виконують роль клієнтів інших машин, а клієнти можуть експортувати свої ресурси на сервер.

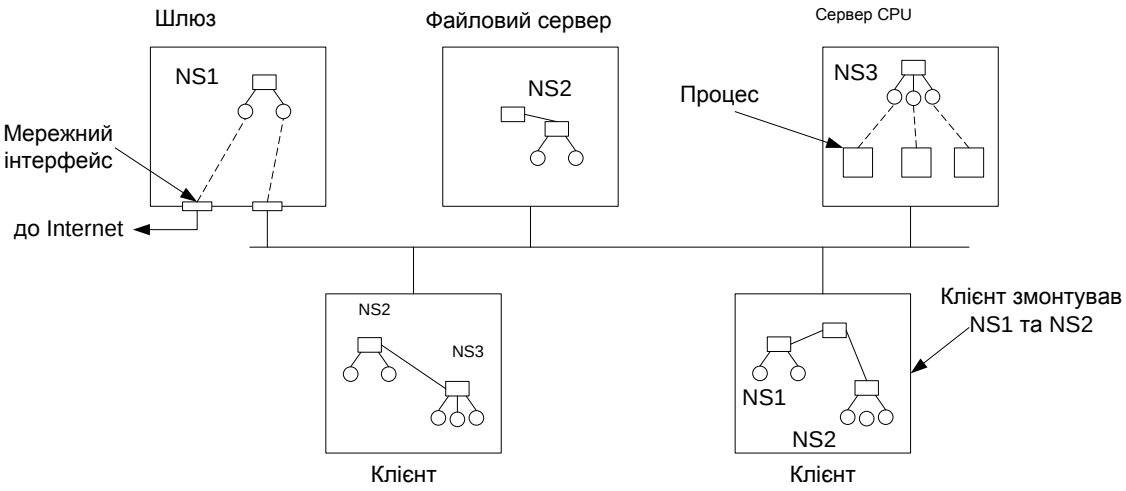


Рис. 5.14. Загальна реалізація Plan9

Система Plan9 грунтується на трьох основних принципах:

- усі ресурси подані як файли і доступні в ієрархічній файловій системі;
- локальні й віддалені ресурси не розрізняються, тому для доступу до них реалізований стандартний протокол 9P;
- кожна група процесів має власний простір імен, зібраний із файлових ієрархій, наданих різними ресурсами.

На відміну від сучасних операційних систем, у яких користувач отримує доступ до персонального комп'ютера або робочої станції, у Plan9 користувач отримує доступ до розподіленого обчислювального

середовища і має можливість конфігурувати свій робочий простір. Так, */dev/mouse* для процесу описує пристрій «миша» комп'ютера, з якого цей процес запущено, причому це може бути необов’язково комп'ютер, на якому виконується процес.

У системі Plan9 є багато незвичайних серверів із файловими інтерфейсами. Віконна система Rio надає користувачеві можливість працювати з текст-орієнтованим графічним середовищем (терміналом, клавіатурою, мишею тощо), показаним на рис. 5.15. Програми можуть здійснювати введення/виведення тексту через пристрій */dev/cons*, подання графіки через */dev/draw*, отримувати події з пристрою «миша», читаючи командою */dev/mouse* тощо.

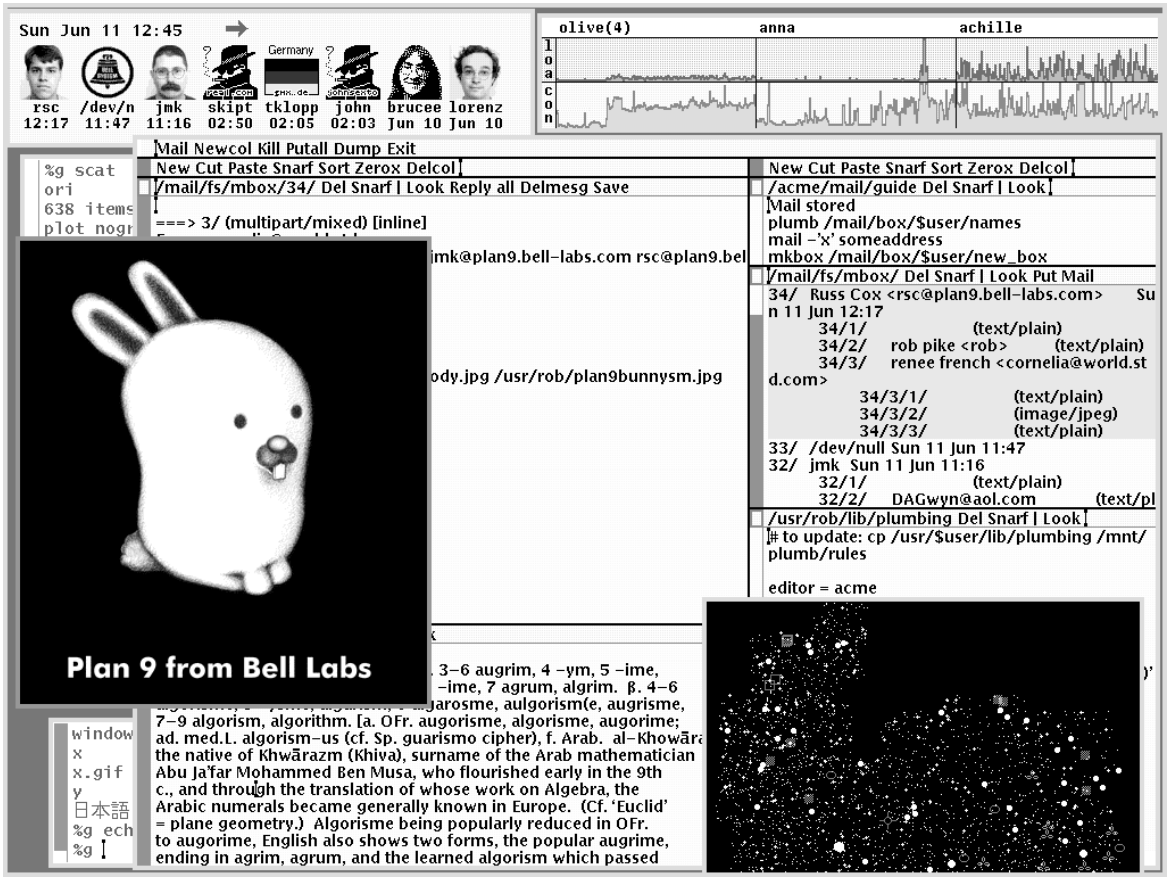


Рис. 5.15. Інтерфейс розподіленої файлової системи Plan9

Таблиця 5.2. Основні повідомлення, які використовуються в системі

Повідомлення	Дія
version	Узгодження версій протоколу
error	Повернення помилки
flush	Переривання повідомлення
auth, attach	Повідомлення для встановлення з'єднання
walk	Зміна каталогу, рух деревом каталогів
create, open	Підготовка обробника для проведення операцій над файлом
read, write	Передача даних із файлу або у файл
clunk	Закриття обробника
remove	Видалення файлу із сервера
stat, wstat	Запит атрибутів файлу на їх змінювання

5.6. xFS – файлова система без серверів

xFS – високопродуктивна розподілена ФС, створена компанією SiliconGraphics для власної операційної системи IRIX, яка відрізняється від інших ФС тим, що вона спочатку була розрахована для використання на дисках великого об'єму (більше 2 Терабайт). Система xFS підтримує журналізацію, тобто здійснює ведення журнала, який зберігає список змін, що допомогає зберігати цілісність файлової системи під час збоїв.

Підтримку xFS була включено в ядро Linux версій 2.4 (починаючи з версії 2.4.25). Інсталятори дистрибутивів SuSE, Gentoo, Mandriva, Slackware, Ubuntu, Fedora та Debian пропонують xFS як варіант файлової системи для встановлення. FreeBSD також стала підтримувати xFS у режимі читання у 2005 році.

xFS має архітектуру без серверів: вся ФС розподілена по багатьом клієнтським машинам. Такий підхід різко контрастує з архітектурою більшості інших файлових систем, які мають жорстко централізовану

структуру, навіть якщо в них для розподілу і реплікації файлів використовується не один, а декілька серверів. До подібних систем відносяться AFS і Coda.

До XFS поставлено такі вимоги:

- швидке відновлення після збоїв;
- підтримка великих розділів і файлів;
- ефективна робота з великими каталогами;
- висока масштабованість як за продуктивністю, так і за обсягами сховищ;
- ефективна протидія фрагментації;
- високий ступінь паралельності обробки запитів.

Систему xFS було розроблено для локальних мереж, у яких машини сполучені між собою високошвидкісними каналами. Цим вимогам задовольняє множина наявних мереж, наприклад внутрішні мережі кластерів робочих станцій. Для повного розосередження даних і функцій керування машинами локальної мережі розробники xFS прагнули досягти вищих масштабованості й відмовостійкості, ніж у традиційних розподілених файлових системах із файловими серверами. Прототип xFS працював на робочих станціях Sun SPARC і ULTRASPARC, сполучених мережею Myrinet.

В архітектурі xFS виокремлюють три типи процесів.

Сервер зберігання (storage server) – це процес, який відповідає за зберігання частин файлу. Спільно сервери зберігання реалізують масив віртуальних дисків, аналогічний реалізації дискових масивів RAID (Redundant array of independent(or inexpensive) disks).

Менеджер метаданих (metadata manager) – це процес, який відстежує, де насправді зберігаються блоки файлів. Відзначимо, що блоки даних одного файлу можуть бути розкидані по декількох серверах зберігання. Менеджер передає запити клієнтів відповідним

серверам зберігання, тобто діє подібно до сервера локалізації для блоків даних, з яких складаються файли.

Клієнт (client) у xFS – це процес, який отримує запити користувачів на виконання операцій із файлами. Кожен клієнт має можливість кешування і може передавати кешовані дані іншим клієнтам.

Основний підхід, покладений в основу xFS під час проектування цієї системи, полягає в тому, що ролі клієнтів, менеджерів і серверів можуть виконувати будь-які машини. У повністю симетричній системі на кожній машині виконуються всі три процеси, але можна також використовувати віддалені машини, на яких функціонуватимуть тільки сервери зберігання, тоді як на інших машинах наявні лише процеси клієнтів або менеджерів, що показано на рис. 5.16.

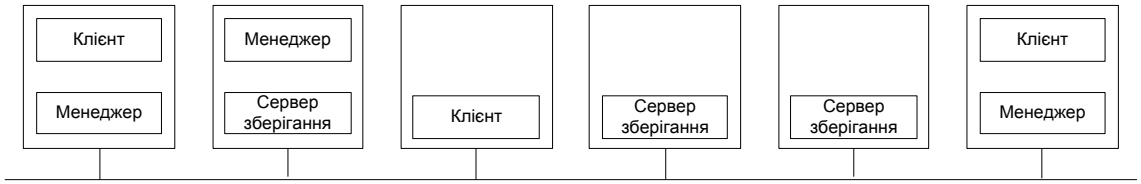


Рис. 5.16. Типовий розподіл процесів xFS по декількох машинах

Структуру розподіленої ФС xFS показано на рис. 5.17.

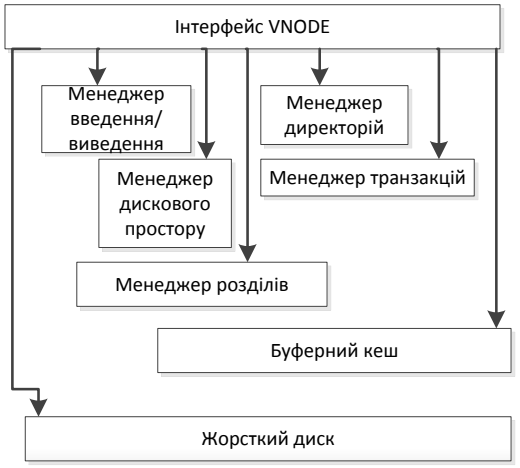


Рис. 5.17. Архітектура розподіленої файлової системи xFS

Узагальнена структура xFS подібна звичайній файловій системі, в яку додано менеджер транзакцій (TransactionManager) і менеджер розділів (Volume Manager). xFS повністю підтримує всі файлові

інтерфейси UNIX і відповідає стандартам POSIX і XPG4. Вона розміщується нижче від інтерфейсу VNODE в ядрі операційної системи IRIX і використовує весь спектр сервісів ядра, включаючи виведення/сторінковий кеш (буфер/кеш сторінки), кеш пошуку імен у каталозі, динамічний кеш vnode.

xFS охоплює кілька модулів, кожен з яких реалізує певну частину її функціональності. Основна і найбільш важлива частина файлової системи – менеджер дискового простору (SpaceManager). Цей модуль керує вільним місцем у ФС, розміщенням дескрипторів і виділенням дискового простору конкретним файлам. Менеджер введення-виведення (I/O Manager) відповідає за виконання файлових запитів на введення-виведення і звертається до менеджера дискового простору за здійсненям процесів виділення і відстеження дискового простору для файлів. Менеджер директорій (Directory Manager) реалізує простір імен файлової системи. Буферний кеш використовують всі частини xFS для кешування в пам'яті вмісту найбільш часто запитуваних блоків зазначеного розділу. Менеджер транзакцій використовують інші частини файлової системи, щоб зробити всі операції зміни метаданих xFS атомарними, дозволяючи швидко відновити цілісність файлової системи після збоїв. Volume Manager (XLV), який використовується в xFS, реалізує шар абстракції дискової файлової системи та виконує всі дискові операції читання, запису і відображення, запитувані файловою системою. Сама xFS нічого не знає про розміщення і геометрії тих пристроїв, на яких вона працює. Відділення дискових операцій від основного коду файлової системи значно спростило її реалізацію і використання.

Особливості розподіленої ФС xFS такі:

- 64-бітна файлова система;
- ведення журналу тільки для метаданих;
- зміна розміру в реальному режимі часу (тільки збільшення);

− дефрагментація в реальному режимі часу;

− запис на диск здійснюється лише у разі нестачі пам'яті, що дозволяє зменшити фрагментацію та знизити активність запитів до диску;

− інструменти резервного копіювання і відновлення (*xfsdump* і *xfsrestore*).

Недоліки xFS такі:

− неможливість зменшити розмір наявної файлової системи;

− відновлення видалених файлів у xFS – дуже складний процес, тому нині є лише кілька програмних продуктів для відновлення видалених файлів із цієї файлової системи;

− можливість втрати даних під час запису у разі збою живлення, оскільки велика кількість буферів зберігається у пам'яті;

− порівняно високе навантаження на центральний процесор;

− до останніх версій на 32-розрядних системах індексні блоки можуть розміщуватися лише в початкових 2-терабайтах диску.

Результати тестів показують, що розподілена ФС xFS – найкраща для застосування, якщо потрібні маніпуляції з великими файлами.

5.7. Secure File System

У захищеній файловій системі (Secure File System, SFS) основною особливістю є відділення засобів керування ключами від засобів захисту файлової системи, тобто SFS гарантує, що клієнт не зможе дістати доступ до файлу, не володіючи відповідним секретним ключем, отримання якого абсолютно не залежить від того, до якого файлу відбувається доступ.

Загальну структуру SFS подано на рис. 5.18, яка для забезпечення сумісності системи з різноманітними машинами виконана у вигляді декількох компонентів системи NFS. На машині клієнта, крім програми

користувача, розміщуються три різні компоненти системи. Клієнт NFS використовується як інтерфейс із програмою користувача і обмінюється інформацією з клієнтом SFS, котрий представляється клієнтові NFS як сервер NFS, тобто ці два компоненти обмінюються інформацією за допомою RPC протоколом NFS.

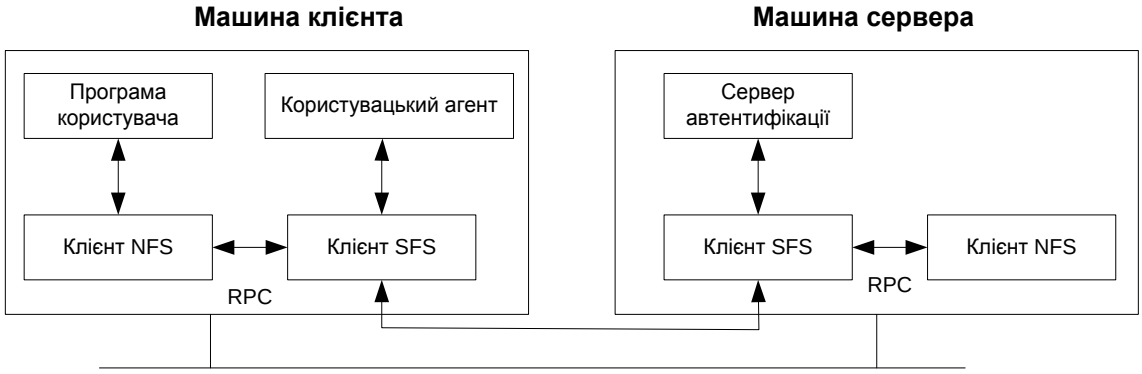


Рис. 5.18. Структура SFS

Клієнт SFS організує захищений канал з сервером SFS, зв'язує з локальним призначеним для користувача агентом (*user agent*). SFS автоматично ідентифікує користувача і не встановлює способу автентифікації користувачів. Відповідно з принципами захисту, SFS виділяє певний процес в окреме завдання і використовує різних агентів для різних протоколів автентифікації користувача.

На стороні сервера з міркувань мобільності використовується сервер NFS, який підтримує зв'язок із сервером SFS, що для сервера NFS представляється клієнтом. Сервер SFS створює процес ядра SFS, який відповідає за обробку файлових запитів клієнтів SFS. Аналогічно агентові SFS сервер SFS для автентифікації користувачів підтримує зв'язок з виділеним сервером автентифікації.

Особливості розподіленої ФС SFS такі:

− масштабованість захисту – рівень захисту залежить від складності ключа;

− до п'яти шифрованих томів можуть бути доступними в будь-який час;

− працює з більшістю типів дисків, з дискетами обсягом 360 Кб та до 9 Гб SCSI (Small Computer Systems Interface) дисків;

− надає прямий доступ до вбудованих інтерфейсів IDE (Integrated Drive Electronics) та EIDE (Enhanced Integrated Drive Electronics) дисків для підвищення продуктивності;

− SCSI пристрої доступу для підвищення продуктивності й використання дисків, які зазвичай не доступні для DOS;

− всі ділянки пам'яті захищаються для запобігання витоку даних;

− мінімізується можливість виконання інших програм моніторингу або зміни її роботи;

− шифровані томи можуть бути швидко демонтованими обумовленими користувачем клавішами; автоматично демонтованими після певного часу; демонтованими під контролем смарт-карти; перетвореними з зашифрованого в незашифрований вигляд;

− підтримка знімних жорстких дисків та магнітооптичних дисків.

5.8. Розподілена файлова система Ceph

Розробляють систему «Ceph» з листопада 2007 року, але вже з кінця березня 2010 року її використано в основний гілці ядра Linux (починаючи з версії 2.6.34).

Призначення «Ceph» визначено необхідністю отримання таких характеристик функціонування системи:

− легкість масштабування до петабайтних розмірів;

− висока продуктивність під дією різних навантажень (кількості операцій введення/виведення за секунду і різної смуги пропускання);

− підвищена надійність.

Ці характеристики є протилежними, наприклад, масштабованість може знижувати продуктивність або заважати її підвищенню, або впливати на надійність. У системі «Ceph» застосовано кілька

концепцій, зокрема динамічний поділ метаданих і розподіл/реплікація даних. В архітектуру Ceph закладено також стійкість до окремих відмов, причому передбачено, що на великих обсягах даних (петабайт даних) відмови в системі зберігання є звичайним явищем, а не винятковим. Систему не розробляли під конкретне навантаження, тобто вона може адаптуватися до зміни розподіленого навантаження, що забезпечує найкращу продуктивність. Усі функції системи сумісні зі стандартами POSIX, завдяки чому розгортання системи не вплине на наявні програми, які функціонують у межах POSIX (використовуються розширення, пропоновані в Ceph). Ceph є розподіленою системою зберігання даних із відкритим вихідним кодом, яка входить в основну гілку ядра Linux (починаючи з версії 2.6.34).

Оскільки систему «Ceph» розроблено для забезпечення високої продуктивності, надійності й масштабності, то її технічні рішення дозволили подолати дві суттєві проблеми, характерні для розподілених файлових систем:

- вартість системи, Ceph має відкрите програмне забезпечення, яке розповсюджують безкоштовно;
- масштабованість – система придатна для зберігання від гігабайт до петабайт інформації. Десять тисяч клієнтів, або навіть більше, можуть здійснювати одночасно доступ до одного і того ж файлу або директорії.

Ceph можна поділити на чотири частини (рис. 5.19): клієнти (користувачі даних); сервери метаданих, які кешують і синхронізують розподілені метадані; кластер зберігання об'єктів, у якому у вигляді об'єктів зберігаються як дані, так і метадані, та в якому реалізовані інші ключові сутності); кластерні монітори, за допомогою яких реалізовані функції моніторингу.

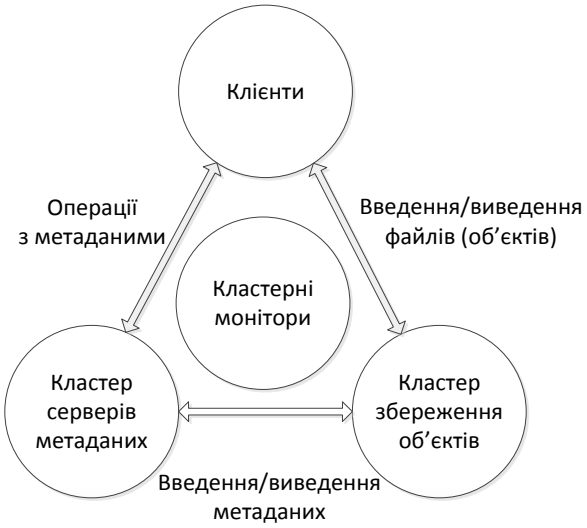


Рис. 5.19. Структура розподіленої файлової системи Ceph

Основні особливості, які відрізняють Ceph від інших систем:

− безшовне масштабування, тобто ФС «Ceph» може бути легко розширена новим пристроєм зберігання даних. Порівняно з іншими системами Ceph відразу переносить частину даних на новий носій для розподілу навантаження і рівномірного використання наявного ресурсу;

− висока надійність і швидке відновлення - вся інформація копіюється на декілька носіїв, і у разі втрати однієї з копій відразу створюється нова;

− адаптивний сервер метаданих - сервер розроблено так, щоб він автоматично підлаштовувався до поточного навантаження. З плином часу кількість звертань і розмір ієрархії файлової системи збільшуються, тоді ця ієрархія динамічно розподіляється між серверами метаданих.

5.9. Файлова система Google File System

Google File System (GFS) – розподілена ФС, яку використовує компанія Google, є однією з найбільш відомих розподілених файлових систем. Надійне масштабоване зберігання даних вкрай необхідно для будь-якого прикладного програмного забезпечення, що працює з таким

великим масивом даних, як усі документи в Internet. Система GFS є основною платформою зберігання інформації в Google, яку розробляли, враховуючи такі критерії:

− систему будують з великої кількості звичайного недорогого обладнання, яке не має високих надійності та відмовостійкості, тому слід здійснювати моніторинг збоїв, і забезпечити можливість у разі відмови якого-небудь устаткування відновлення функціонування системи;

− система має зберігати не тільки багато великих файлів, але й значну кількість маленьких, для яких не оптимізується її робота;

− система має реалізовувати суворо окреслену семантику паралельної роботи декількох клієнтів, у разі, якщо вони одночасно намагаються дописати дані в один і той самий файл. За умови, що надійдуть одночасно сотні запитів на запис в один файл, використовується атомарність операцій додавання даних у файл з деякою синхронізацією. Якщо надійде операція на читання, то вона буде виконуватися або до чергової операції запису, або після неї;

− висока пропускна здатність є кращою, ніж незначна затримка. Так, більшість програм у Google віддають перевагу роботі з великими обсягами даних на високій швидкості, а виконання окремо взятої операції читання і запису може виконатись на протязі більшого періоду часу;

− файли в GFS організовані ієрархічно, за допомогою каталогів, як і в будь-якій іншій файловій системі, та ідентифікуються за допомогою зазначення шляху. З файлами в GFS можна виконувати звичайні операції: створення, видалення, відкриття, закриття, читання і запис;

− система GFS підтримує резервні копії, або знімки (snapshot), копії файлів або дерева директорій, причому з невеликими витратами.

Архітектуру GFS показано на рис. 5.20.

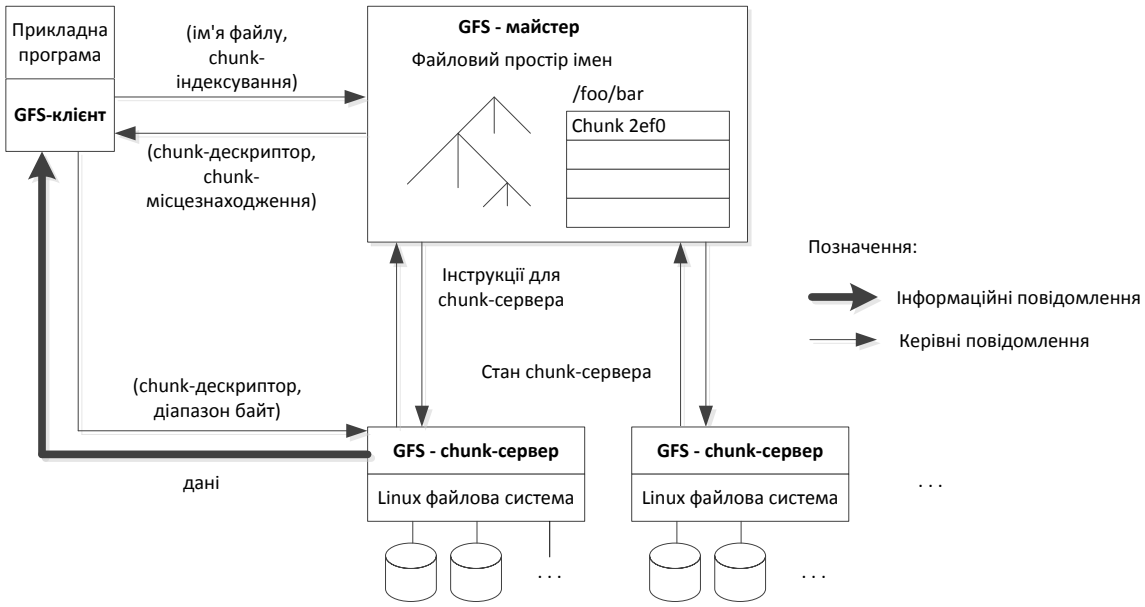


Рис. 5.20. Архітектура GFS

GFS-кластер складається з однієї головної машини майстра (master) і множини машин, що зберігають фрагменти файлів (chunk – фрагмент файла) chunk-сервери (chunk-servers), де власне зберігаються дані. Клієнти мають доступ до всіх цих машин. Файли в GFS розбиваються на фрагменти, які мають фіксований розмір, що налаштовується. Кожен такий фрагмент має унікальний і глобальний 64-бітний ключ, який видає майстер, створюючи його. Chunk-сервери зберігають фрагменти як звичайні Linux-файли на локальному жорсткому диску. Для надійності кожен фрагмент може бути репліковаим на інші chunk-сервери; зазвичай використовуються три репліки. Майстер відповідає за роботу з метаданими всієї файлової системи, які охоплюють простори імен, інформацію про контроль доступу до даних, відображення файлів у фрагменті й поточне положення фрагментів. Майстер контролює всю глобальну діяльність системи також, зокрема керування вільними фрагментами, збирання сміття (непотрібних фрагментів) і їх переміщення між chunk-серверами. Майстер постійно обмінюється повідомленнями (HeartBeat messages) із chunk-серверами, щоб надати інструкції та визначити їх стан (дізнатися, чи ще живі).

Клієнт взаємодіє з майстром тільки для виконання операцій, пов'язаних із метаданими, а у разі розробки прикладного програмного забезпечення не використовують кешування даних, не зважаючи на те, що клієнти керують метаданими, тому, що на chunk-серверах операційна система Linux кешує найбільш використовувані блоки в пам'яті.

Використання одного майстра у складі GFS-кластера істотно спрощує архітектуру системи, дозволяє робити складні переміщення фрагментів, організовувати реплікації, використовуючи глобальні дані.

Клієнти ніколи не читають і не пишуть дані через майстра, а запитують у нього, з яким chunk-сервером їм варто контактувати, а далі вони спілкуються із chunk-серверами безпосередньо.

Розглянемо, як відбувається читання даних клієнтом. Спочатку, знаючи розмір фрагмента, ім'я файлу і зміщення відносно початку файлу, клієнт визначає номер фрагмента всередині файлу. Потім він відправляє майстру запит, який містить ім'я файлу та номер фрагмента в цьому файлі. Майстер видає клієнту ідентифікатор chunk-сервера, по одному в кожній репліці, а також ідентифікатор фрагмента. Потім клієнт вирішує, яка з реплік йому подобається більше (зазвичай та, яка ближче), і надсилає запит, що складається із фрагмента і зміщення відносно його початку; подальше читання даних не вимагає втручання майстра. На практиці клієнт в один запит на читання вводить відразу кілька фрагментів, і майстер дає координати кожного з них у одній відповіді.

Розмір фрагмента є важливою характеристикою системи. Переважно його вважають рівним 64 Кб, що більше, ніж розмір блока у звичайній файловій системі. Зрозуміло, що якщо необхідно зберігати багато файлів, розміри яких менші від розміру фрагмента, то витрачатиметься значний обсяг зайвої пам'яті. Але вибір такого

великого розміру фрагмента обумовлений завданнями, які компанія Google повинна вирішувати на своїх кластерах.

Майстер зберігає такі важливі види метаданих: простір імен файлів і фрагментів, відображення файлу в фрагментах і положення реплік фрагментів. Усі метадані зберігаються в пам'яті майстра, тому операції майстр виконує швидко. Про стан системи майстер дізнається просто й ефективно: він виконує періодичні сканування chunk-серверів у фоновому режимі для визначення застарілих даних, додаткових реплікацій, у разі виявлення недоступного chunk-сервера і переміщення фрагментів, для балансування навантаження і вільного місця на жорстких дисках chunk-серверів.

Взаємодія складових системи. Кожна зміна фрагмента має дублюватися на всіх репліках і змінювати метадані. У системі GFS майстер дає фрагмент у володіння одному із серверів, який зберігає цей фрагмент. Такий сервер називають первинною (primary) реплікою, а решту реплік - вторинними (secondary). Первинна репліка збирає послідовні зміни фрагмента, всі репліки дотримуються цієї послідовності, якщо ці зміни відбуваються. Схему взаємодії складових системи показано на рис. 5.21.

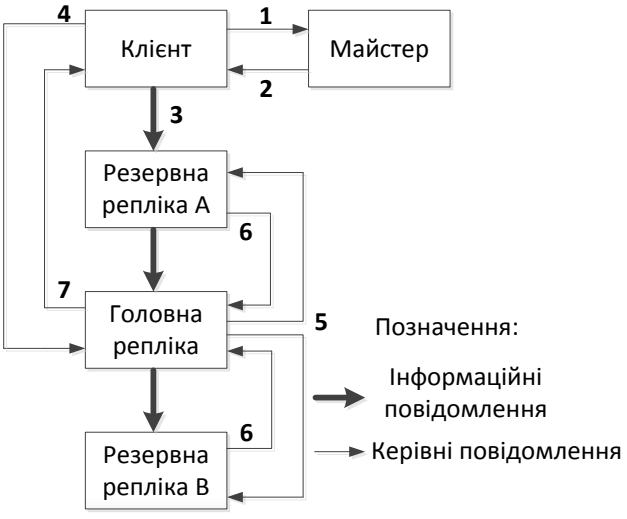


Рис. 5.21. Взаємодія складових системи Google File System

1. Клієнт запитує майстра, який із chunk-серверів володіє фрагментом і де міститься цей фрагмент в інших репліках. Якщо необхідно, то майстер віддає фрагмент комусь у володіння.

2. Майстер у відповідь видає первинну та вторинні репліки, клієнт зберігає ці дані для подальших дій. Тепер спілкування з майстром клієнтові може знадобитися лише тоді, коли первинна репліка стане недоступною.

3. Далі клієнт відсилає дані в усі репліки, причому може це робити в довільному порядку. Кожен chunk-сервер буде їх зберігати у спеціальному буфері, поки вони не знадобляться або не застаріють.

4. Коли всі репліки отримають ці дані, клієнт надсилає запит на запис первинної репліки. У цьому запиті містяться ідентифікація даних, надісланих у кроці 3. Тепер первинна репліка встановлює порядок, у якому мають відбуватися всі зміни, які вона отримала, можливо від декількох клієнтів паралельно. І потім виконує ці зміни локально у встановленому нею порядку.

5. Первинна репліка пересилає запит на запис всім уторинним реплікам, кожна з яких виконує ці зміни в порядку, визначеному первинною реплікою.

6. Вторинні репліки інформують про успішне виконання цих операцій.

7. Первинна репліка надсилає відповідь клієнту, а також помилки, що виникли у будь-якій репліці. Якщо помилка виникла під час записування в первинній репліці, то і записування у вторинні репліки не відбувається. У цьому разі клієнт обробляє помилку і вирішує, що йому робити далі.

З опису процесу взаємодії складових системи зрозуміло, що в ній розділено потік даних і потік керування.

Стійкість до збоїв і діагностика помилок. Часті перебої у роботі компонентів системи є однією з найскладніших проблем. Кількість і

якість компонентів роблять ці збої не винятком, а нормою через надвисоку складність системи. Збій компонента може бути спричинений недоступністю цього компонента або, що гірше, наявністю зіпсованих даних.

GFS підтримує систему в робочому стані за допомогою двох простих стратегій: швидкого відновлення і реплікації.

Швидке відновлення – це, фактично, перезавантаження машини, при цьому час запуску незначний, що зумовлює незначну затримку, а потім робота продовжується. Майстер реплікує фрагмент, якщо одна з реплік стала недоступною або пошкодилися дані, які містять репліку фрагмента. Пошкоджені фрагменти знаходять, обчислюючи контрольні суми.

В системі використовують ще один вид реплікації – реплікацію майстра, коли реплікують реєстр записів про операції (log) та контрольні точки (checkpoints). Кожна зміна файлів у системі відбувається тільки після того, як майстр запише log операцій на диски машин, на які він реплікується. У разі незначних несправностей майстер може здійснити перезавантаження, у разі проблем із жорстким диском або іншою інфраструктурою майстра, без якої він не здатний функціонувати, система GFS стартує нового майстра на одній з машин, на яку необхідно реплікувати дані майстра.

Таким чином, система GFS підтримує великомасштабну обробку даних із навантаженням на стандартних апаратних засобах. У системі несправність компонентів є нормальним явищем завдяки апаратній частині, на якій реалізовано систему. Система забезпечує відмовостійкість за рахунок постійного моніторингу, реплікації важливих даних, а також швидкого й автоматичного відновлення. Конструкція забезпечує високу сумарну пропускну здатність у процесі виконання різних завдань.

5.10.Відкрита розподілена файлова система OpenAFS

Open Andrew File System (OpenAFS) — відкрита реалізація розподіленої файлової системи AFS. Ключовими моментами є незалежність від місця перебування клієнта, прозора міграція, кешування і резервування даних. Усі призначені для користувача файли утворюють єдину ієрархію */afs*, включаючи об’єднані хости (клієнти і сервери) для ефективного сумісного доступу до файлів через локальну мережу (local aria network, LAN) і глобальну мережу (wide aria network, WAN). Клієнти кешують часто використовувані об'єкти (файли) для швидкого доступу до них.

Система AFS грунтується на розподіленій файловій системі, деякі вже існуючі розробки називали свої файлові системи як AFS, щоб уберегти ранні AFS сайти від перейменування їх файлових систем. Це дозволило застосувати AFS як самостійну систему та зберегти ім'я і корінь (кореневий каталог) розподіленої файлової системи.

Середовище AFS – це група об'єднаних адміністративно серверів, що являє собою єдину файлову систему. Зазвичай, ядро AFS – це набір хостів, які мають однакове доменне ім'я (наприклад, gentoo.org). Користувач під'єднується до робочої станції AFSclient, яка вимагає інформацію та файли від імені користувача. Користувачі не будуть знати, на якому сервері міститься файл, який вони запитують, файли завжди доступні. Схему взаємодії елементів OpenAFS показано на рис. 5.22.

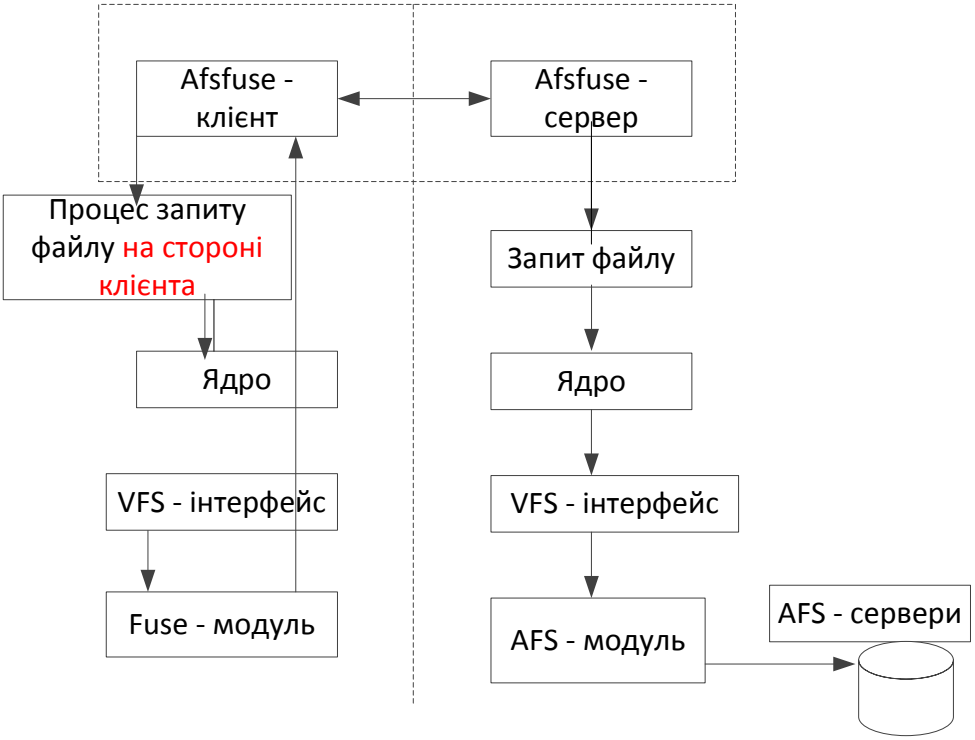


Рис. 5.22. Схема взаємодії елементів розподіленої файлової системи OpenAFS

В основу алгоритму функціонування ФС OpenAFS покладено два процеси – на стороні сервера (Afsfuse-server) та на стороні клієнта (Afsfuse-client), де FUSE (*FilesysteminUserspace*) – модуль, який застосовують в ядерах Unix-подібних операційних систем.

Основні переваги системи AFS – це можливість кешування (на стороні клієнта кешують, зазвичай, від 100Mb до 1Gb), безпека (використовує технологію Kerberos та списки контролю доступу (access control lists)), простота адресації, масштабованість, тобто здатність додавати сервери за потребою.

5.11.Протокол WebDAV

Web-based Distributed Authoring and Versioning (WebDAV) – сучасний і захищений мережний протокол високого рівня, що працює поверх HTTP для доступу до об'єктів і їх колекцій.

Особливості використання WebDAV такі:

– виконання основних файлових операцій з об'єктами на віддаленому сервері;

– виконання розширених файлових операцій (блокування, підтримка версій);

– робота з будь-яким типом об'єктів (не лише із файлами);

– підтримка метаданих (властивостей) об'єктів;

– підтримка одночасної роботи з кількома об'єктами.

Основні сценарії застосування протоколу WebDAV, які дозволяють створити саме його особливості використання, такі:

– спільна робота з web-документами;

– робота користувачів з мережною файловою системою;

– розподілена розробка програмного забезпечення;

– уніфікований доступ до сховища даних.

Схему взаємодії користувача з елементами мережі з використанням WebDAV показано на рис. 5.23.

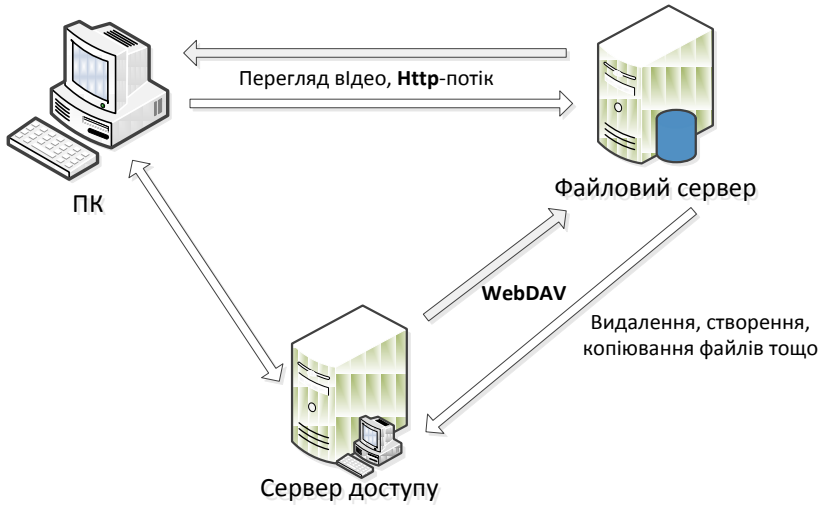
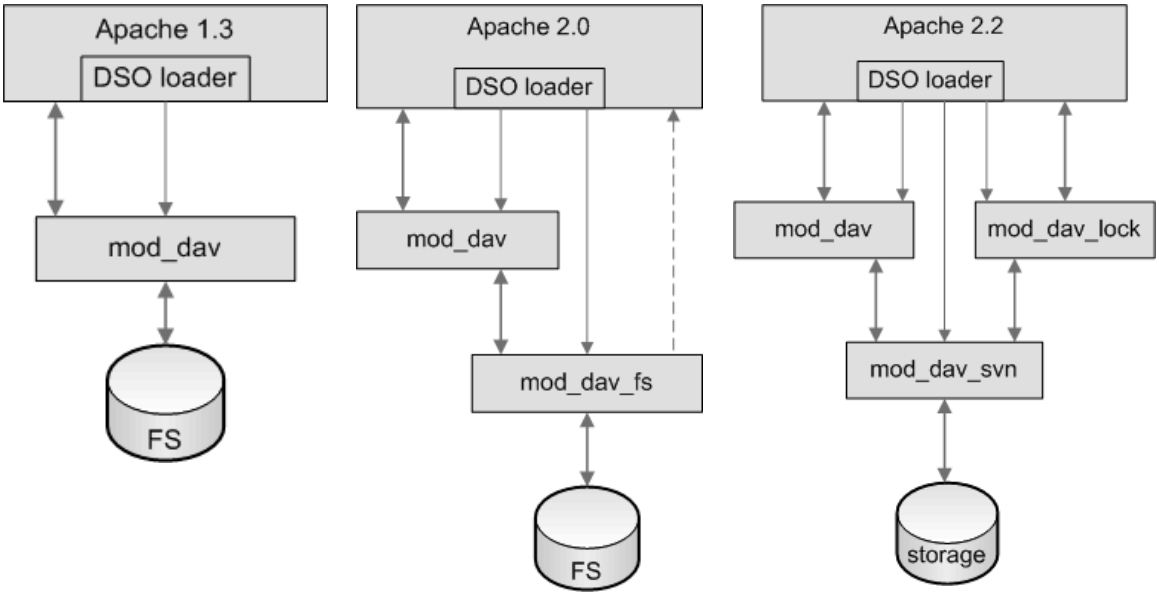


Рис. 5.23. Взаємодія користувача з елементами мережі з використанням WebDAV

WebDAV реалізовано на основі http-сервера Apache, побудова WebDAV залежна від застосовуваної версії Apache (рис. 5.24), а сам WebDAV створюють у вигляді модуля, який міститься на сервері доступу.



а) *б)* *в)*

Рис. 5.24. Реалізація WebDAV модулем на apache-серверах різних версій

У версії apache 1.3 (рис. 5.24, *а*) як сховище ресурсів і колекцій використовується файлова система сервера. Реалізація WebDAV для Apache 2.0 (рис. 5.24, *б*) значно відрізняється від версії 1.3, тому, що розробники *mod_dav* вирішили дати можливість адміністраторам самим обирати, що використовувати як сховище ресурсів, розділивши всі функції WebDAV на такі дві групи:

1. Група функцій, яка відповідає за обробку протоколу WebDAV і реалізує взаємодію з уніфікованим сховищем, цю групу реалізовано в модулі *mod_dav*.

2. Група функцій, які призначені для функціонування сховища ресурсів, цю групу реалізовано у модулі *mod_dav_fs*, де як сховище ресурсів застосовано файлову систему.

Таким чином, якщо необхідно використовувати яке-небудь нестандартне сховище ресурсів WebDAV, то треба лише створити відповідний модуль, який буде застосовувати інтерфейс прикладного програмування (Application Programming Interface, API) модуля *mod_dav*.

У разі використання сервера Apache 2.2 (рис. 5.24, *в*) підвищено гнучкість модуля *mod_dav* – в окремий модуль *mod_dav_lock* винесено

API для роботи з блокуваннями, який з'явився в Apache 2.1. Модуль *mod_dav_fs* не потребує обов’язкової наявності модуля *mod_dav_lock*, оскільки використовує власну реалізацію бази блокувань.

Приклади застосування WebDAV такі:

- Subversion - WebDAV для спільної роботи під час розробки програмного забезпечення;
- Microsoft Explorer/ Office/ Outlook - протокол WebDAV використовується для можливості працювати з документами на віддаленому сервері;
- Adobe GoLive - протокол використовується для спільної роботи з web-документами;
- Apache - реалізація підтримки WebDAV в модулі *mod_dav*;
- Zope і Tomcat - WebDAV використовує для керування контентом.

У цілому WebDAV розширив протокол HTTP кількома новими заголовками (*DAV: / If: / Depth: / Overwrite: / Destination: / Lock-Token: / Timeout: / Status-URI:*) і методами (*PROPFIND; PROPPATCH; KCOL; LOCK /UNLOCK; COPY /MOVE*).

5.12.Порівняльна характеристика розподілених файлових систем

Цілі розробки файлових систем зазвичай різноманітні. Файлова система NFS – це система, яка надає клієнтам прозорий доступ до файлової системи конкретного віддаленого сервера, є подібною до систем об'єктів, які підтримують лише віддалені об'єкти. Розробники NFS версії 4 також намагалися реалізувати прозорий доступ до глобальних мереж, унеможлививши затримки на передавання даних за рахунок повного кешування файлів на стороні клієнта й локального виконання всіх операцій.

Основна мета розробки Coda полягала в забезпеченні доступності навіть у разі поривів у мережі й роботі в автономному режимі. Система Coda також відповідає цілям створення AFS, найбільш важливою з яких є масштабованість. Поєднання високої доступності й масштабованості вирізняє Coda серед більшості інших розподілених файлових систем.

Система Plan9 передусім являє собою розподілену систему розділення часу, де доступ до всіх ресурсів здійснюється однаково, а саме як доступ до файлів. З погляду на це систему Plan 9 можна також назвати системою розподілу файлів.

Система xFS націлена, як і багато інших розподілених файлових систем, на забезпечення високої доступності та масштабованості. Відмінністю є те, що доступність і масштабованість в xFS досягаються в умовах відсутності виділених серверів.

Систему SFS призначено для масштабування системи захисту за рахунок роздільного вирішення питань керування файловою системою та її захисту. Будь-якому користувачеві дозволяється запустити власну службу SFS, не вносячи змін до роботи централізованих служб, важливою складовою SFS є організація простору імен – до імені файлу додається відкритий ключ сервера, на якому міститься файл.

Систему Ceph проектували для можливості зберігати надвелику кількість інформації, розробивши алгоритми адаптації серверів до динамічного навантаження.

Однією з найперспективніших і ефективно працюючих систем є GFS, в основу якої покладено принципову відмінність від інших файлових систем – архітектуру виконано на основі великої кількості звичайного недорогого обладнання, яке може спричиняти перебої в роботі, але завдяки реалізованому резервуванню така система працює надзвичайно ефективно.

Систему OpenAFS виконано на основному принципі побудови розподілених ФС, який полягає у тому, що доступ до файлів не залежить від місця розташування й перебування клієнта. Цей принцип реалізовано як для LAN, так і для WAN-мереж.

Відмінністю WebDAV є те, що це - сучасний і захищений мережний протокол високого рівня для доступу до об'єктів і колекцій об'єктів.

Наявні й інші важливі відмінності, наприклад у базових моделях файлів, підтримуваних кожною з систем. Так, NFS, Plan 9 і SFS підтримують модель віддаленого доступу, в якій за файли постійно відповідає сервер, що здійснює всі операції з файлами. Coda до певної міри підтримує модель завантаження/вивантаження через кешування файлів, яке ця система успадкувала від AFS. У xFS використовується абсолютно інший підхід, за якого реалізація файлової системи має структуру журналу.

Для більш детального порівняння розподілених файлових систем потрібно визначити множину критерів порівняння та грунтовно вивчити кожну з них на практиці.

5.13. Висновки

1. Сучасні розподілені файлові системи, які мають більш розвинену функціональність, зокрема OpenAFS, Coda, GFS та WebDAV, мають більш високу швидкодію, покращений захист, додаткові можливості керування розділами, а також створення резервних копій. Розподілені ФС є однією з найважливіших парадигм розподілених систем завдяки побудові у відповідності з моделлю клієнт-сервер із кешуванням на стороні клієнта і підтримкою реплікації серверів для отримання необхідної масштабованості. Кешування і реплікація потрібні для

підвищення доступності цих систем і реалізовані у більшості розподілених ФС.

2. Розподілені ФС відрізняються від нерозподілених семантикою розділення файлів. Ідельні ФС завжди дозволяють клієнтові прочитати дані, записані у файл останніми. Таку семантику розділення файлів (семантику UNIX) дуже важко ефективно реалізувати в розподілених системах.

3. Система NFS підтримує її «скорочений» варіант - семантику сеансів, за якої остаточну версію файлу визначає останній клієнт, що закрив файл, відкритий для запису.

4. У Coda розділення файлів відбувається відповідно до семантики транзакцій, тобто клієнти, що виконують читання, отримають останню версію файлу в разі лише повторного його відкриття. Семантика транзакцій у Coda не має всіх властивостей ACID, які притаманні справжнім транзакціям. Якщо керування всіма операціями здійснює сервер, то може використовуватися й базова семантика UNIX, хоча масштабованість у цьому разі залишиться неповною.

5. Щоб досягти прийнятної продуктивності, розподілені ФС зазвичай дозволяють клієнтам кешувати файли цілком, що підтримується в системах NFS і Coda, хоча існує можливість зберігати також і великі фрагменти файлів. Після того, як файл відкрито і передано (частково) клієнтові, решта операцій може здійснюватися локально. Зміни скидаються на сервер перед закриттям файлу.

6. Системи Plan 9 і xFS підтримують блокове кешування в комбінації з протоколом кеша зворотного запису.

7. На практиці розподілена ФС не створюється поверх транспортного рівня, а використовує наявний рівень RPC, так що всіма операціями є виклики RPC, адресовані файловому серверу, і потреби в передаванні повідомлень не виникає. Такий підхід використовується в NFS, Coda, SFS та інших системах. Бажано, щоб рівень RPC

підтримував семантику звертань «максимум одного разу», інакше ця семантика має бути реалізована у формі частини рівня файлової системи, як у NFS.

8. Coda відрізняється від інших розподілених файлових систем тим, що підтримує роботу в автономному (від'єднаному від мережі) режимі. У такому разі, якщо гарантувати, що кеш клієнта завжди містить дані, які знадобитимуться йому найближчим часом, можна дозволити клієнтові продовжувати роботу, навіть якщо він тимчасово від'єднаний від файлового сервера. Для підтримки такого режиму роботи необхідне спеціальне керування кешем, яке називають накопиченням.

9. Захист дуже важливий для будь-яких розподілених систем, зокрема файлових. Система NFS на відміну від Coda і Plan 9 не має яких-небудь механізмів захисту, але, володіючи стандартизованим інтерфейсом, дозволяє використовувати різноманітні системи захисту, наприклад Kerberos. Автентифікація в цих системах здійснюється за допомогою захищених викликів RPC і часто аналогічна протоколу автентифікації Нідхема – Шредера, натомість у SFS інформацію про відкритий ключ сервера внесено до імен файлів.

5.14.Запитання для самоконтролю

1. Що таке NFS?
2. Які моделі функціонування наявні в системі NFS?
3. Що таке CODA?
4. Яку систему взято за основу в CODA?
5. Які процеси використовуються в системі CODA?
6. Яка роль процесу Venus у системі CODA?
7. Що таке xFS?
8. Яка основна особливість системи xFS?
9. Які наявні процеси в системі xFS?

10. Що таке SFS?

11. Яким чином досягається безпека у разі роботи з системою SFS?

12. Що таке Plan 9?

13. Чому систему Plan 9 також називають системою розподілу файлів?

14. Що таке Ceph?

15. Які три основні переваги Ceph порівняно з іншими системами?

16. Що таке GFS? Де реалізована система GFS?

17. Які критерії обрано для побудови GFS?

18. Поясніть принцип роботи GFS.

19. Який порядок взаємодії клієнт-сервера в системі GFS?

20. Назвіть переваги GFS серед інших розподілених ФС.

21. Що таке OpenAFS? Поясніть, як працює система OpenAFS.

22. Що таке WebDAV?

23. Наведіть приклади застосування системи WebDAV.

24. Поясніть принцип взаємодії користувачів з елементами мережі у роботі WebDAV.

25. Назвіть відмінності реалізації WebDAV з різними версіями Apache.

6. РОЗПОДІЛЕНІ СИСТЕМИ ДОКУМЕНТІВ

Однією з найважливіших причин, що сприяли широкому застосуванню розподілених систем у всіх сферах діяльності людства, стала поява Всесвітньої павутини (World Wide Web, WWW). Перевага технології Web полягає у порівняній простоті парадигми: все навколо – це документи, тому Web нині є найважливішою розподіленою системою документів. Іншою важливою системою документів, що з'явилася раніше є Lotus Notes, в основу якої, на відміну від Web, покладено не файли, а бази даних.

6.1. World Wide Web

World Wide Web можна вважати гігантською розподіленою системою, в якій для доступу до пов'язаних між собою документів функціонують мільйони клієнтів і серверів. Сервери підтримують набори документів, а клієнти надають користувачам простий інтерфейс для доступу і перегляду цих документів. Кожний документ міститься у файлі (хоча в деяких випадках документи можуть генеруватися за запитом). Сервер отримує запити на видачу документів, а також на збереження нових документів, і передає їх клієнту.

Найпростіший спосіб посилання на документ – уніфікований ідентифікатор ресурсу (Uniform Resource Locator, URL). URL-адреса подібна міжопераційному посиланню на об'єкт (IOR) у CORBA або контактній адресі у Globe та визначає, де розміщується цільовий документ, за рахунок додавання до початкового документу DNS-імені відповідного сервера разом з ім'ям файлу, за допомогою якого сервер знаходить цільовий документ у своїй локальній файловій системі. Крім того, URL визначає протокол прикладного рівня серед декількох наявних, що використовується для пересилання документа мережею.

Клієнт взаємодіє з web-серверами за допомогою спеціальної програми, яку називають браузером (browser), що відповідає за правильне відображення документа, обробляє операції введення від користувача, основна з яких - вибір посилання на інший документ, який потім витягується зі сховища і виводиться на екран. Ця схема відповідає узагальненій структурі Web, поданій на рис. 6.1.

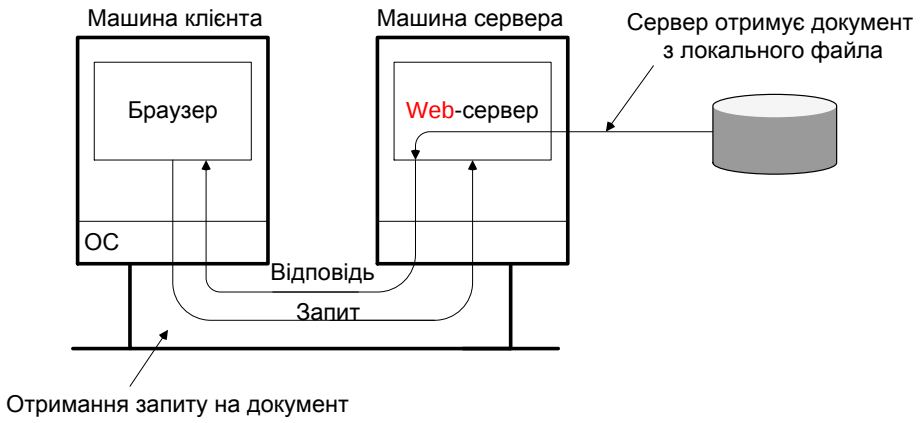


Рис. 6.1. Узагальнена структура Web

6.1.1. Документна модель

Web грунтується на поданні всієї інформації у вигляді документів, для створення яких є багато способів. Деякі документи можуть бути простими текстовими ASCII-файлами, інші - набором сценаріїв, які автоматично виконуються у процесі завантаження документа у браузер.

Документ може містити посилання на інші документи, які називають гіперпосиланнями (hyperlinks). Коли документ відкривається у браузері, гіперпосилання виділяються, щоб користувач легко міг їх помітити й обрати посилання, клацнувши на ньому мишею. Вибір гіперпосилання генерує запит на отримання документа, який пересилається на сервер, на якому зберігається документ. Звідти він повертається на машину користувача і відкривається у браузері. Новий документ може вікритися в новому вікні або замінити наявний документ у поточному вікні браузера.

Більшість документів у Web описується за допомогою особливої мови розмітки гіпертексту HTML. Термін «розмітка» означає, що HTML містить ключові слова для розбиття тексту документа на окремі розділи.

Щоб уможливити перенесення даних документу, модель дерева синтаксичного розбору слід стандартизувати. Так стандарт вимагає, кожен з вузлів має містити лише елементи одного типу з визначеного в ньому набору типів елементів та реалізовувати стандартний інтерфейс, що містить методи доступу до його вмісту, повертати посилання на батьківські й дочірні вузли тощо. Це стандартне подання відоме як об'єктна модель документа (Document Object Model, DOM), яку часто називають динамічною мовою HTML (Dynamic HTML).

Нині значна кількість web-документів пишеться мовою HTML, але все більш широкого розповсюдження набуває інша узгоджена з DOM мова – розширювана мова розмітки (Extensible Markup Language, XML), яка, на відміну від HTML, забезпечує лише структуризацію документа, оскільки не містить ключових слів для форматування документа, наприклад вирівнювання тексту по центру або подання його курсивом, й може використовуватися для визначення довільних структур, надаючи засоби визначення різних типів документів.

Подання XML-документа користувачу вимагає визначення правил його форматування, для чого застосовують найпростіший підхід – вбудування XML-документу в HTML-документ, використовуючи для форматування ключові слова HTML.

Крім того, для створення web-документів можна застосувати спеціальну мову форматування, розширену мова стилів (Extensible Style Language, XSL).

6.1.2. Типи документів

Крім HTML- і XML-документів, розрізняють велику кількість інших типів документів, наприклад, сценарії, документи у форматах PostScript або PDF, зображення у форматах JPEG або GIF, звукові документи у форматі MPx, документи у вигляді типу багатоцільового розширення пошти Інтернету (Multipurpose Internet Mail Extensions, MIME), який розроблено для передачі інформації у вигляді вмісту тіла повідомлень, що становлять частину електронної пошти та застосовується у WWW. У MIME розрізняють типи повідомлень за їх вмістом, наявні типи верхнього рівня і підтипи. Різні типи верхнього рівня містять текст, зображення, аудіо і відео. Спеціальний тип *Application* вказує, що документ містить дані, які стосуються певної програми, яка здатна перетворити документ у документ, який зможе сприйняти людина.

6.1.3. Огляд архітектури

За рахунок комбінування HTML або XML із сценаріями утворюється певний засіб відображення документів, для функціонування якого необхідне середовище WWW, створене спочатку з відносно простих систем з архітектурою клієнт-сервер (рис. 6.1), яку було розширено за рахунок множини компонентів, які підтримують ускладнені типи документів.

Одним з перших доповнень базової архітектури стала підтримка нескладної взаємодії з користувачами за допомогою узагальненого інтерфейсу шлюзів CGI, який визначає стандартний спосіб виконання web-сервером програм із даними користувача як вхідними параметрами. Зазвичай дані користувача вводяться в HTML-форму, яка визначає програму, яка має виконуватися на стороні сервера, і значення

параметрів, уведених користувачем. Після того як заповнення форми закінчено, ім'я програми і зібрані значення параметрів пересилаються на сервер, як це показано на рис. 6.2.

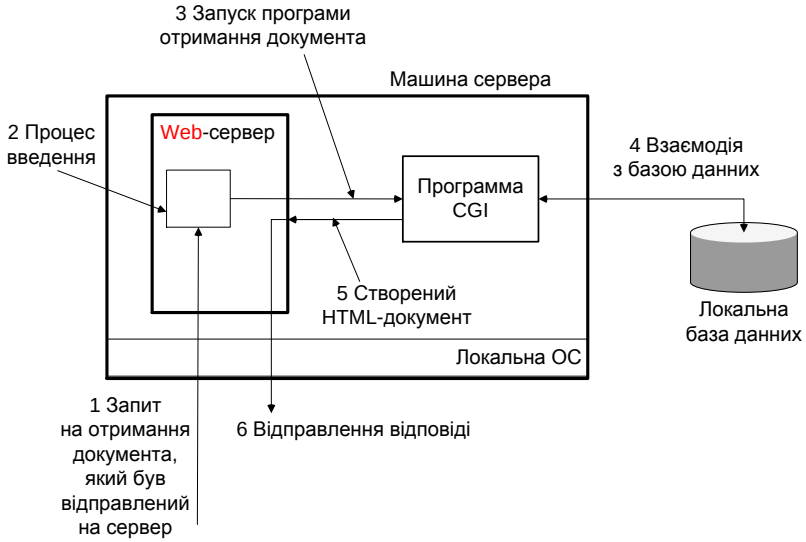


Рис. 6.2. Принцип використання програм CGI на стороні сервера, 1,2,3,4,5,6 – порядок виконання кроків під час виконання запиту

Коли сервер знаходить запит, він запускає вказану в запиті програму і передає їй параметри, які містяться у запиті, після чого програма виконує обробку даних і повертає результати у вигляді документа, який відправляється назад браузеру клієнта, щоб клієнт міг його побачити.

Основне завдання сервера, який використовується для обробки запитів клієнта, полягає у отриманні документа. За наявності програм CGI отримання документа може бути делеговано цим програмам, так що сервер не знає про те, чи був документ створений «на льоту», чи його дійсно одержали з локальної файлової системи.

Одне з найважливіших удосконалень в обробці документів полягає в тому, що сервери можуть обробляти отримані документи перед їх відправленням до клієнта. Зокрема, документ може містити серверний сценарій (server-side script), який виконує сервер під час локального отримання документа. Результати виконання сценарію разом з іншими частинами документа пересилаються клієнту, проте сам сценарій не

пересилається. Інакше кажучи, серверний сценарій підмінює документ, замінюючи сценарій результатом його виконання.

Окрім виконання клієнтських і серверних сценаріїв, клієнту можна також передавати наперед скомпільовані програми у вигляді аплетів (applet), тобто невеликих автономних прикладних програм, які можна відіслати клієнту і виконати у просторі адрес браузера. Найчастіше використовують аплети у вигляді програм мовою Java, скомпільованих у інтерпретований Java-код. Java-аплет можна інтегрувати в HTML-документ.

Аплети зазвичай виконуються на стороні клієнта, але застосовують також і серверні варіанти аплетів, які називають сервлетами (servlets). Як і аплети, сервлети - це наперед скомпільовані програми, які виконуються в адресному просторі серверу та на практиці пишуть переважно мовою Java, але жодних фундаментальних обмежень щодо використання інших мов немає. Сервлет реалізує методи, які є також в HTTP – стандартному комунікаційному протоколі взаємолдії між клієнтом і сервером.

Коли документ одержить сервер, у разі якщо йому потрібна додаткова обробка, активуються серверні сценарії, які містяться в документі та виконуються на сервері. На практиці серверні сценарії потрібні лише для документів, які були безпосередньо отримані з локальної файлової системи, тобто документів, одержаних без допомоги сервлетів або програм CGI. Після завершення обробки документ пересилається браузеру користувача.

Після того як документ одержить клієнт, браузер виконує клієнтські сценарії і у разі потреби активує і виконує аплети, які містить документ. Результати обробки документа у вигляді його вмісту виводяться на термінал користувача (рис. 6.3)

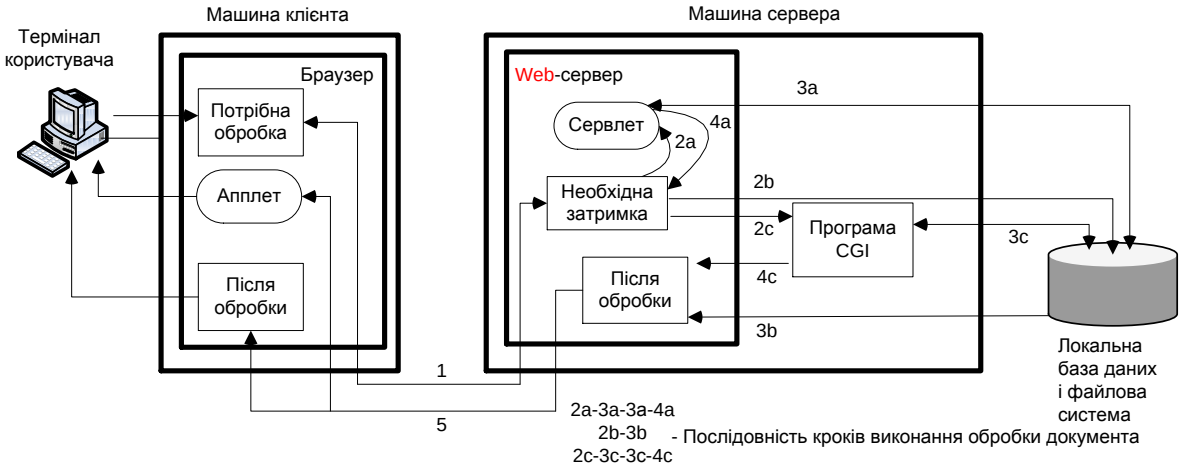


Рис. 6.3. Архітектура клієнт - сервер у Web

В описаній архітектурі передбачено, що браузер може показувати HTML-документи і XML-документи, але для документів в інших форматах (PostScript або PDF) браузер завантажує документ із файлової системи сервера без жодної обробки. В результаті такого підходу клієнт одержує файл. Використовуючи розширення файлу, браузер запускає відповідну прикладну програму, яка дозволяє продемонструвати або обробити його вміст. Оскільки такі прикладні програми допомагають браузеру показувати документи, їх називають також прикладними програмами-помічниками (helper applications).

6.2. Lotus Notes

Інша розподілену система Lotus Notes орієнтована на бази даних. Її розробиа корпорація Lotus Development, але нині продаж і розробку цієї системи здійснює компанія IBM. Система працює під керуванням різних платформ сімейств Windows і UNIX.

Як і Web, Lotus Notes є системою з архітектурою клієнт-сервер, яку спочатку розроблено для роботи в локальних мережах, але на сьогодні система може працювати і в глобальних мережах та в Internet. Загальну структуру системи показано на рис. 6.4, яка містить чотири основні компоненти: клієнти, сервери, бази даних і програмне забезпечення проміжного рівня. Кожний клієнт або сервер може мати

декілька локальних баз даних, кожна база даних формує колекцію записів (notes), які є ключовими елементами даних у будь-якій системі Lotus Notes.

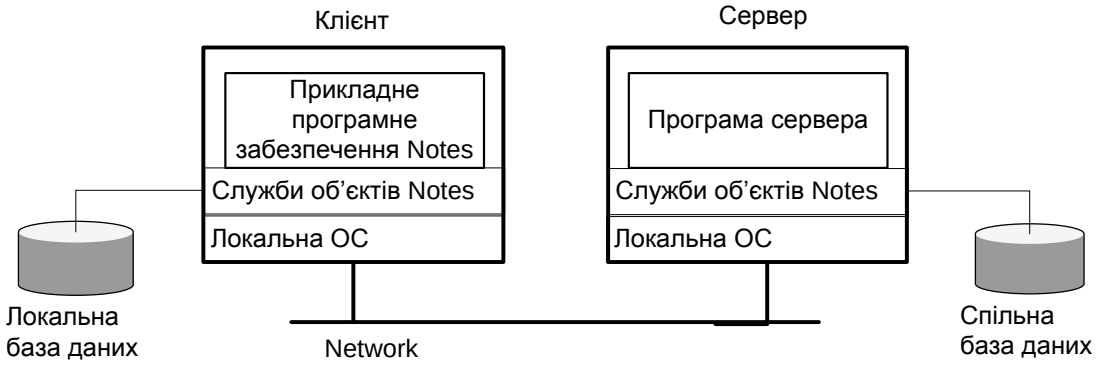


Рис. 6.4. Загальна структура системи Lotus Notes

Сервери Lotus Notes називають Domino (Domino servers), які керують колекцією баз даних, тобто їх завдання – надавати доступ до цих баз даних віддаленим клієнтам й іншим серверам. Основна програма сервера складається з модулів, які чекають запитів, що надходять мережею, підтримують з'єднання і сеанси з віддаленими процесами, обробляють інформацію, яка міститься в локальних базах даних.

Документи у Web зазвичай реалізуються у вигляді файлів, натомість Lotus Notes зберігає та обробляє документи за допомогою технологій баз даних. Це головна й дуже важлива відмінність цих двох систем, яка і визначає різницю між web-серверами та серверами Domino.

Клієнти, сервери і бази даних поєднуються за допомогою окремого компонента проміжного рівня, який називають службами об'єктів Notes (Notes Object Services, NOS). Цей компонент реалізується поверх базових операційних систем і мереж, дозволяючи клієнтам і серверам зв'язуватися між собою і використовувати локальні бази даних за допомогою компонентів віддаленого виклику процедур, засобів зберігання даних, механізмів зворотного виклику тощо.

Документна модель. Основний елемент даних у системі Lotus Notes – це запис (note) або список елементів, в якому елемент (item) – це комірка, в якій зберігаються дані, відповідні запису. Кожний елемент має тип, який визначає, якого роду дані можуть у ньому зберігатися. Залежно від цього розрізняють: текстові елементи для зберігання простого тексту, числові - для чисел з плаваючою точкою, елементи - для дати і часу, елементи формул - для скомпільованих команд Notes, елементи сценарію - для сценаріїв та ін.

Кожен запис може мати список асоційованих із ним дочірніх записів та лише одного асоційованого з ним батька, утворюючи ієрархію записів, яка відображає одну зі специфічних цілей систем Notes, а саме створення такої системи, яка дозволила б користувачам надсилати записи, а потім відповідати на них. Зв'язок між первинним записом і відповіддю на нього підтримується у вигляді співвідношення батько-дитина майже так само як у системах мережних новин.

У системі записи поділяють на записи з даними (data notes) та інші типи записів. Запис із даними або документ можна порівняти з web-документом: він відтворює документ, який містить різні елементи даних – аудіо- і відеозаписи, зображення, текст, значки тощо.

Множину інших типів записів можна умовно розділити на записи конструкції, які використовують для подання документів і роботи з ними, та записи адміністрування, які необхідні для керування базами даних Notes.

Запис Form є аналогом форми, яку застосовують у Web або у традиційних базах даних та яка визначає, як має виглядати документ, включаючи положення і вигляд його елементів, які стають полями форми, при цьому деякі поля можуть спільно використовуватися різними формами. Ця можливість задається за допомогою запису *Field*. Запис *View*, який реалізує подання, визначає, як має бути показана користувачу колекція документів, наприклад, як таблиця.

Розрізняють й інші записи, зокрема запис ACL, який використовують для зберігання списку контролю доступу, запис ReplFormula, який є прикладом запису адміністрування та визначає як відбуватиметься реплікація баз даних в Notes.

Як і документи у Web, записи можуть посилатися один на одного за допомогою гіперпосилань, які в Notes називають посиланнями на посилання (notelinks). Посилання на запис визначає базу даних і запис, який міститься в цій базі, тобто допускається застосування перехресних посилань на записи з різних баз даних. Можна також посилатися на запис, використовуючи ідентифікатори URL, які зазвичай вбудовуються в посилання на запис разом із посиланням на сервер, який керує відповідною базою даних.

Не зважаючи на те, що записи у Lotus Notes подібні web-документам, вони побудовані абсолютно інакше, зокрема їх розділено на зміст запису, який визначається його елементами, і спосіб подання користувачам, який визначається елементами конструкції.

6.3. Порівняння World Wide Web і Lotus Notes

Розглянемо порівняння систем World Wide Web і Lotus Notes за такими критеріями: основні принципи побудови систем, організація зв’язку, процеси обробки документів, призначення імен, синхронізація, кешування і реплікація, відмовостійкість, захист.

Основні принципи побудови систем. Системи World Wide Web (WWW) і Lotus Notes використовують просту модель клієнт-сервер, у якій сервер керує набором розкиданих багатьма серверами документів, доступних віддаленим клієнтам, але суттєво відрізняються їх моделі документів.

Web-документ використовує модель, в якій документ є текстовим файлом, що містить різноманітні команди розмітки мовою HTML або

однією з інших мов (наприклад, XML). Проте ці документи можуть містити і документи інших типів, зокрема зображення, аудіо- і відеофрагменти. Крім того, у міру зростання популярності Web з'явилася можливість вбудовувати в документ складніші елементи, наприклад сценарії та аплети.

Оскільки Web продовжує розвиватися, HTML як мова розмітки поступово замінюється мовою XML, яка дозволяє краще описати структуру документа. Крім того, XML відділяє структуру документа від його подання, тоді як у HTML вони поєднані.

Модель, яка лежить в основі Lotus Notes, суттєво відрізняється від моделі, обраної у Web. Основним елементом даних у Notes є список елементів або запис, на відміну від якого у Web записом є структура даних, прив'язана до бази даних. Зміна і відображення запису здійснюється виключно механізмами баз даних, такими як форми і подання, які, у свою чергу, формують за допомогою спеціальних записів, тобто елементів конструкції.

Інша відмінність між Web і Lotus Notes полягає в тому, що в Notes за допомогою записів здійснюється керування всією системою, тому наявні спеціальні записи для керування реплікацією, забезпечення захисту тощо. Натомість, у Web для цієї мети використовуються окремі механізми.

Зв'язок. Взаємодія у Web відбувається за спеціальним протоколом HTTP, який визначає всі допустимі операції з документами, які обмежуються отриманням документа із сервера і переміщенням його на сервер.

Взаємодія в Lotus Notes підтримується за допомогою традиційної підсистеми RPC – віддаленого виклику процедур. Крім того, в Notes застосовують механізми зв'язку верхнього рівня, передусім електронну пошту, для чого в цій системі передбачені різноманітні засоби автоматичного приймання, обробки і відправлення пошти.

Слід зазначити, що взаємодія між процесами на одній машині також здійснюється по-різному: у Web поведінка клієнтів і серверів жорстко визначається базовою операційною системою, тобто будь-який зв'язок між процесами реалізують механізми операційної системи. У Notes для маскування відмінностей між операційними системами клієнтів і серверів використовується спеціальний рівень, NOS, що покращує переносимість багатьох прикладних програм Notes.

Процеси обробки документів. Порівнюючи Web і Lotus Notes за процесами, які підтримують життєвий цикл документів, їх обробку, бачимо, що багато завдань вирішуються в них однаково. Найважливішими клієнтами для Web є браузери, які надають користувачам графічний інтерфейс для отримання і огляду документів. У Lotus Notes клієнту також надається графічний інтерфейс для огляду записів, що зберігаються у віддаленій базі даних, а також спеціальні програми для редагування записів за допомогою форм, подання тощо. Оскільки клієнт Notes призначений для роботи з усіма типами записів (наперед визначеними), потреби в наданні йому додаткової функціональності не виникає.

Організація серверів цих двох систем також має схожі риси за винятком того, що традиційний web-сервер зазвичай більш пристосований для роботи із файловою системою, тоді як сервер Notes Domino ближчий до традиційних серверів баз даних.

Гнучкість серверної частини Web реалізується за допомогою програм CGI, які переважно запускаються у вигляді окремого процесу і можуть взаємодіяти з базами даних. Таким чином, можливості web-серверу наближаються до можливостей сервера Domino. Крім того, web-сервери можуть підтримувати модулі, які динамічно завантажуються, і відомі як сервлети.

Призначення імен. Відмінність у системах призначення імен між Web і Notes пов'язана із організацією Web на основі файлів, що

обумовлює іменування документів за допомогою URN або URL, і використанням баз даних у Notes. Вказівники URL зазвичай містять ім'я, яке відповідає локальному імені файлу документа, збереженого на сервері, натомість імена URN призначені для використання як ідентифікатори, які не залежать від місцезнаходження.

У Notes документи (й інші типи записів) мають пов'язаний з ними універсальний ідентифікатор, який використовується для іменованого запису і доступу до нього. Для пошуку записів у базі даних немає потреби в осмисленому (з погляду людини) ідентифікаторі, який утворюється з рядка символів. Notes має інтерфейс іменування для Web, який побудовано на основі вказівника URL і містить універсальний ідентифікатор записів, а також імена сервера і бази даних, у яких зберігається вказаний запис.

Синхронізація. Підтримка синхронізації в Web, і в Notes мінімальна, тобто зводиться до локальних механізмів блокування, а блокування декількох документів, розміщених на різних серверах, немає. Лише останнім часом з'явилися пропозиції для Web щодо підтримки колективного редагування з блокуванням документів, використовуваних спільно.

Кешування і реплікація. У результаті порівняння засобів кешування і реплікації з’ясовано, що між Web і Notes є істотні відмінності.

Кешування відіграє у Web особливу роль, бо підвищує масштабованість усієї системи, натомість у Notes документи використовують спільно і часто змінюють, тому застосовано механізми кешування баз даних, які є менш гнучкими, ніж у Web. Нині традиційно використовують ієрархічні схеми кешування, в яких кешування здійснюється на рівні користувачів, організацій, регіонів і країн. Кешування ефективне тому, що більшість документів у Web дуже рідко змінюється, проте Web-документ також стає динамічним,

що робить кешування, особливо ієрархічне кешування, все менш ефективним.

Для Web було запропоновано і реалізовано різні протоколи підтримки несуперечливості кешу, які підтримують схеми слабкої несуперечливості. Кешований документ уважається правильним доти, поки не закінчиться термін його блокування, за цей час документ буде змінений, але кешовані копії не одержать ніякої інформації про цю подію.

Реплікація у Web традиційно підтримується у формі створення дзеркал для сайту в цілому, проте цей механізм дуже негнучкий. З появою мереж поширення вмісту (Content Delivery Network, CDN) почали застосовувати динамічну реплікацію. Мережі CDN дозволяють динамічно реплікувати документи на серверах, розташованих поблизу користувача, що забезпечує несуперечливість реплік завдяки технологіям CDN.

Відмовостійкість. Для забезпечення відмовостійкості Web і Notes використовують схожі методики. У Web надійність системи зв'язку забезпечує виключно протокол TCP, а відмовостійкість системи - кластери web-серверів, які підвищують доступність і збільшують продуктивність кожного web-сервера, який входить у кластер. Більшість широко застосуваних реалізацій таких кластерів заснована на використанні виділеного сервера зовнішнього інтерфейсу, який передає запити на один із серверів кластера.

У Notes використовуються спеціальні механізми підтримки кластерів серверів Domino з реплікацією баз даних за достатньо строгим протоколом несуперечливості, згідно з яким зміни кожної секунди розсилаються всім серверам кластера та який фактично є ідентичним протоколам реплікації зі слабкою формою поширення змін.

Захист. У Web використовують засоби захисту транспортного рівня (Transport Layer Security, TLS), які дозволяють організувати

захищений канал між клієнтом і сервером, а потім у процесі сеансу зв'язку забезпечувати контроль доступу, а авторизацію здійснює сервер змін.

У Notes використовуються сертифікати автентифікації, при цьому суттєве значення має перевірка достовірності сертифікатів. Для того, щоб вирішити, чи можна довіряти відкритому ключу, який міститься в сертифікаті, в системі підтримується специфічна модель довіри, яка використовує ієрархічну схему іменування та частково конфігурується у процесі настроювання сервера. За допомогою перехресних сертифікатів дві різні системи Notes можуть висловити одна одній взаємну довіру, що дозволяє їх клієнтам використовувати сервери кожної з цих систем.

Керування доступом у Notes реалізовано за допомогою спеціальних записів, які містять записи списку контролю доступу (Access Control List, ACL), що дає змогу надавати найрізноманітніші права доступу разом із різними рівнями доступу.

Порівняння World Wide Web і Lotus Notes подано у табл. 6.1.

Таблиця 6.1. Схожі ознаки й відмінності World Wide Web і Lotus Notes

Аспект	**WWW**	**Lotus Notes**
Базова модель	Гіпертекст	Список текстових елементів (приміток)
Розширення	Мультимедіа, сценарії	Мультимедіа, сценарії
Модель зберігання	На основі файлів	На основі бази даних
Мережна взаємодія	HTTP	RPC, електронна пошта
Взаємодія між процесами	Визначається ОС	Служби NOS
Клієнтський процес	Браузер, редактор	Браузер, редактор конструкції
Клієнтські розширення	Модулі розширення	Надаються ОС клієнта

Закінчення табл. 6.1

Аспект	**WWW**	**Lotus Notes**
Серверний процес	Нагадує файловий сервер	Нагадує сервер баз даних
Серверні розширення	Сервлети, програми CGI	Завдання сервера
Аспект	WWW	Lotus Notes
Кластери серверів	Прозорі	Непрозорі
Іменування	URN.URL	URL, ідентифікатори
Синхронізація	Переважно локальна	Переважно локальна
Кешування	Розширене	Не документовано
Реплікація	Створення дзеркал, CDN	Слабка
Відмовостійкість	Надійний зв’язок і кластери	Кластери
Відновлення	Явної підтримки немає	Одиничного сервера
Автентифікація	Переважно TLS	Перевірка автентичності сертифікатів
Контроль доступу	Залежить від сервера	Розширений, списки ACL

6.4. Розподілені системи документів на підприємствах

Сучасні підприємства для обміну документами використовують єдиний інформаційний простір, у якому і функціонує система поширення документів. Етапи розвитку системи документообігу на сучасному підприємстві, починаючи від автоматизації паперового документообігу і закінчуючи міжвідомчим електронним документообігом, подано на рис. 6.5.

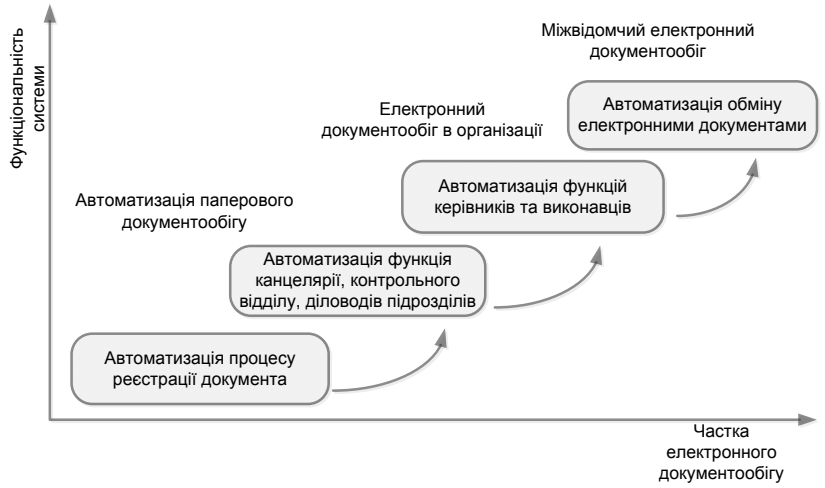


Рис. 6.5. Етапи розвитку системи документообігу

Вимоги щодо доступності документів і потребі у спільній роботі з ними підвищуються експоненційно. Інформаційні матеріали, які мають високу цінність, створюються щодня, розміщуються у глобальних мережах і поширюються в різних професійних колективах. У системі Web формалізовані документи, які були доступні лише фахівцям, не можуть більше служити сховищем корпоративних знань.

Сучасні підприємства вимагають територіально розподіленої архітектури керування документами, тобто такої, яка задовольняє таким вимогам:

− масштабованість, надійність і керованість для економічного корпоративного розгортання;

− автоматична підтримка розподіленого керування різними інформаційними матеріалами впродовж усього їх життєвого циклу: від створення до рецензування, затвердження, поширення й архівації;

− гнучкість керування доступом до всього спектра документів: від електронної пошти до дискусійних баз даних, від відеокліпів до формалізованих документів усіх типів;

− можливість забезпечення миттєвого доступу до документів через web-браузери, настільні прикладні програми й інші загальнодоступні типи клієнтів;

− відкрита, розширювана архітектура, яка дозволяє організаціям, по-перше, швидко розширювати платформу керування документами у зв’язку з появою нових бізнес-цілей, які визначають стратегію керування записами, і, по-друге, інтегрувати керування документами із більш вагомішими стратегічними ініціативами, зокрема керування знаннями;

− доступність широкого спектра додаткових технологій для підвищення рівня повернення від інвестицій;

− розподілене, розширюване керування документами зумовлює суттєве підвищення продуктивності праці співробітників, посилення загальної конкурентоспроможності організації, забезпечуючи оптимізацію будь-якої кількості міждисциплінарних процесів, замість автоматизації окремих вертикальних. Корпоративне керування документами є істотним кроком на шляху до втілення в життя ініціатив з керування корпоративними знаннями.

Розрізняють шість категорій технологій, які становлять ринок засобів електронного керування документами (ЕКД).

Слід зауважити, що жодна класифікація не є ідеальною, через те, що деякі продукти одночасно потрапляють у декілька категорій і мають можливості, характерні для продуктів із різних категорій. Крім того, провідні компанії цього ринку постійно доповнюють функціонал своїх продуктів. Наприклад, Lotus Domino.Doc справедливо віднесено до категорії корпоративних систем ЕКД, хоча він має великі можливості щодо маршрутизації робіт (workflow), особливо у поєднанні з Domino Workflow.

Категорії технологій ЕКД з прикладами найбільш відомих постачальників і продуктів у кожному класі такі:

− системи ЕКД, орієнтовані на бізнес-процеси (Business-process Electronic Document Management, EDM): Documentum, FileNet (Panagon і Watermark), Hummingbird (PC DOCS);

− корпоративні системи ЕКД (Enterprise-centric EDM): Lotus (Domino.Doc), доповнення до Novell GroupWise, Opent Text (LiveLink), Keyfile Corp., Oracle (Context);

− системи керування контентом (Content management): Adobe, Excalibur;

− системи керування інформацією (портали) (Information Management): Excalibur, Oracle Context, PC DOCS/Fulcrum, Verity, Lotus (Domino/Notes, K-station);

− системи керування образами (Imaging);

− системи керування потоками робіт (Workflow management): Lotus (Domino/Notes і Domino Worflow), Jetform, FileNet, Action Technologies, Staffware.

Системи ЕКД, орієнтовані на бізнес-процеси призначені для специфічного програмного забезпечення, враховують особливості документообігу певної галузі або індустрії та забезпечують повний життєвий цикл роботи з документами, включаючи технології роботи з образами, керування записами і потоками робіт, керування контентом тощо.

Корпоративні системи ЕКД забезпечують корпоративну інфраструктуру для створення, спільної роботи з документами та їх публікації, доступну всім користувачам в організації. Основні можливості цих систем аналогічні системам, орієнтованим на бізнес-процеси, проте їх важливою особливістю є спосіб використання і поширення. Корпоративні системи ЕКД аналогічні таким засобам, як текстові редактори й електронні таблиці, є стандартними прикладними програмами за замовчанням для створення і публікації документів у організації. Ці засоби не орієнтовані на використання лише в якійсь певній індустрії або для вузько визначеного завдання, а пропонуються та впроваджуються як загальнокорпоративні технології, доступні фактично будь-якій категорії користувачів.

Системи керування контентом забезпечують процес відстеження створення, доступу, контролю і поширення інформації на рівні розділів документів і об'єктів для їх подальшого повторного використання і компіляції. Доступність інформації не у вигляді документів, а в менших об'єктах полегшує процес обміну інформацією між прикладними програмами.

Системи керування інформацією, або портали, забезпечують керування і поширення інформації мережами Internet, Intranet та Extranet, є основою для створення інформаційних порталів. Системи керування інформацією дають можливість організаціям накопичувати дані та використовувати їх у розподіленому корпоративному середовищі, застосовуючи бізнес-правила, контекст і метадані. Більшість доступних сьогодні технологій забезпечують статичні публікації, тому надання користувачеві вищого рівня інтерактивності та засобів спільної роботи в глобальному середовищі є головним напрямом розвитку систем керування інформацією.

Системи керування образами перетворюють інформацію з паперових носіїв у цифровий формат, зокрема система Tagged Image File Format (TIFF), після чого документ можна використовувати у роботі вже в електронному вигляді.

Системи керування потоками робіт (workflow) забезпечують систематичну маршрутизацію робіт будь-якого типу в межах структурованих і неструктурованих бізнес-процесів для прискорення виконання бізнес-процесів, підвищення їх ефективності й ступеня контролю процесів у організації.

Найпоширеніші системи автоматизації діловодства та документообігу українських, російських та зарубіжних виробників такі:

1. Система **«Megapolis.Документообіг»** від компанії Softline – українського розробника систем електронного документообігу, має підсистему «Підготовка документів», яка дозволяє створювати проекти

документів із шаблонів, здійснювати їх паралельне або послідовне узгодження, створювати типові маршрути узгодження документів, засвідчувати документи електронним цифровим підписом. «Megapolis.Документообіг» – одна з небагатьох в Україні систем, інтегрованих із цифровим підписом із посиленим сертифікатом, який надає юридичної чинності підписаним у системі документам. Використання підсистеми «Підготовка документів» у організації дозволяє до мінімуму скоротити зазвичай затяжний і трудомісткий процес погодження і затвердження документів. Для доступу до розширених можливостей системи «Megapolis.Документообіг» через web-інтерфейс було розроблено спеціалізований web-компонент. Ще однією перевагою продукту є інтеграція з платформою бізнес-аналізу Microsoft Reporting Services, яка дозволяє створювати традиційні та інтерактивні звіти.

2. **КРОН** – система керування документами, призначена для автоматизації діловодства в корпорації, яка має достатньо складну розгалужену структуру й охоплює кілька підприємств, нараховує більше сотні співробітників і має інтенсивний документообіг вхідних і вихідних документів, а також обмін документами між власними підрозділами.

3. Корпоративна система **«Кодекс: Документообіг»** – це комплекс взаємопов’язаних систем діловодства, банків документів і корпоративних сервісів, які забезпечують автоматизоване вирішення завдань діловодства і документообігу в органах державної влади й інших організаціях.

4. **DocsVision 2.0 "Архів-Діловодство"** – це прикладне програмне забезпечення для створення архівів документів, автоматизації основних процедур діловодства і бізнес-процесів обробки документів у організації. Така система може використовуватися як повністю готове рішення для впровадження в компаніях, або як прототип під час

розробки прикладного програмного забезпечення системи документообігу на замовлення. Структуру системи можна налаштувати згідно з потребами конкретної організації.

5. **CompanyMedia -** це Web-орієнтована система керування електронними документами, порівняно недорога, але достатньо функціональна система, яка призначена для виконання таких функцій:

− організації корпоративного сховища документів (каталогу нормативної документації, архіву документів, інформаційного порталу компанії);

− забезпечення процесу колективної підготовки документів, централізованого зберігання робочих документів і автоматичного сповіщення співробітників про документи, що надійшли до опрацювання;

− побудови інформаційного порталу організації, тобто єдиного інформаційного простору, в якому співробітники можуть моментально знайти документи, що їх цікавлять, взяти участь у підготовці й погодженні документів або залучити до цього процесу інших співробітників.

6. **Docs Fusion і Docs Open** від компанії Hummingbird – одна з найпопулярніших у світі систем, які належать до класу «електронних архівів». На жаль, різні покоління і компоненти продукту отримали різні назви, тому у процесі ознайомлення з ним виникає певна плутанина. Спочатку існувала система Docs Open – клієнт-серверна web-орієнтована прикладна програмна система з «товстим» клієнтом, потім було розроблено сервер web-орієнтованого прикладного програмного забезпечення Docs Fusion, який має два клієнта (Windows-клієнт PowerDocs і Web-клієнт CyberDocs) та не вимагає «товстого» клієнта, щоб звертатися безпосередньо до бази даних.

7. **Documentum** – це система керування документами, знаннями і бізнес-процесами для великих підприємств і організацій, розроблена

компанією «Документум Сервисиз». Documentum – це платформа більшою мірою ніж готовий продукт, призначена для створення розподілених архівів, підтримки стандартів якості, керування проектами в розподілених проектних групах, організації корпоративного діловодства, динамічного керування вмістом корпоративних інтранет-порталів.

8. **LanDocs,** передусім орієнтована на діловодство й архівне збереження документів, складається з таких компонентів: системи діловодства, сервера документів (архіву), підсистеми сканування і візуалізації зображень, підсистеми організації віддаленого доступу з використанням Internet-клієнта, поштового сервера.

9. Система **Microsoft SharePoint Portal Server -** електронний архів із розвинутими засобами підтримки спільної роботи; продукт компанії Microsoft, який може претендувати на роль корпоративного та підтримує виконання таких функцій: спільне створення документів, ведення версій документів, вилучення і повернення документів до архіву (check-out, check-in). Для доступу до архіву замість windows-клієнта застосовано web-клієнт і компонент, інтегрований в Windows Explorer, що дозволяє звертатися до архіву як до набору файлів.

10. Система **Optima Workflow** – це більше ніж workflow-продукт, оскільки, крім загального механізму організації потоку робіт, дозволяє зберігати на час виконання робіт усі документи, що стосуються процесу, використовуючи як сховище механізм спільних папок Microsoft Exchange. Превагою цієї системи є відслідковування критичних шляхів і подання комплексу взаємопов’язаних робіт у вигляді діаграм Ганта.

11. Система **«БОСС-Референт»**, яка розроблена компанією АйТі, . належить до категорії систем, орієнтованих на підтримку керування організацією, ефективної роботи співробітників і на накопичення знань, маючи розвинені додаткові сервіси. Основне застосування –

створення корпоративної системи документообігу на підприємстві, що охоплює діяльність співробітників на своїх робочих місцях і підтримує такі бізнес-процеси: діловодство, організаційне управління, погодження документів. Система використовує специфічні поняття, зокрема ролі, функції, які властиві організаціям зі складною ієрархічною структурою. Відмінною рисою системи «БОСС-Референт» є реалізація функцій (Customer Relationship Management, CRM) CRM-системи: контролю договорів, обліку матеріальних цінностей, потокового сканування і розпізнавання, електронної конференції та дошки оголошень.

12. Система **«Дело»** є системою автоматизації діловодства, яка набула значної популярності завдяки послідовній підтримці всіх правил діловодства. Система розроблена компанією «Електронні офісні системи» та підтримує ідеологію діловодства, суть якої полягає в тому, що для виконання будь-якої дії в організації потрібен документ, рух якого забезпечується за рахунок змінювання облікових записів про документ у базі даних.

13. Систему **«Євфрат»,** простий електронний архів із базовими можливостями контролю виконання, розробила компанія Cognitive Technologies, яка пропонує спектр продуктів для організацій різних масштабів – від версії системи для малого офісу до системи для великих компаній («Євфрат Клієнт-сервер», у якому як клієнтська частина використовується «Євфрат-Офіс», що є самостійним продуктом, який може працювати незалежно від серверного компонента системи).

14. Система **Company Media**, розроблена російською компанією «Інтертраст» на основі Lotus Notes, містить широкий набір сервісів, що підтримують діловодство, колективне створення документів, контроль виконання, керування договорами, проектами, управління персоналом, облік матеріальних цінностей тощо. Перевагою системи є ефективна

підтримка територіально розподілених структур керування за рахунок спеціальних методів, які гарантують поширення завдань незалежно від якості ліній передавання. Систему можуть широко застосовувати в організації – і як базу для автоматизації діловодства, і як засіб підтримки роботи співробітників у організації загалом.

15. **Lotus Domino.doc** - прикладне програмне забезпечення для Notes/Domino, розроблено компанією Lotus, яка має достатньо розвинений електронний архів, який дозволяє в середовищі Notes реалізувати корпоративне сховище документів, забезпечує функції збереження версій, контроль вилучення і повернення документів (check-out, check-in), а також, будучи доповненою компонентом Domino Workflow, допомагає реалізувати потоки робіт.

16. Система **Staffware**, яка належить до категорії workflow-систем масштабу підприємства та є серверною технологією для керування потоками робіт, розробила однойменна компанія. Типовими користувачами Staffware можуть стати телекомунікаційні компанії, великі й середні банки, готелі, інші організації, які щоденно виконують множину регламентованих типових операцій.

17. **«Эфект-Офіс»** – система петербурзької компанії «Гарант Інтернешнл», який містить електронний архів, засоби опису структури організації, обмеження доступу за рольовим принципом і маршрутизацію документів. Основна функція системи – електронний архів із засобами пошуку інформації. Крім того, він містить засоби автоматизації діловодства, які ґрунтуються на технологіях маршрутизації документів контролю виконання. У цьому продукті реалізовано власну електронну пошту з підтримкою протоколів POP3/SMTP і UUCP.

6.5. Вимоги до систем документів

6.5.1. Загальні вимоги до систем документів

Загальні вимоги до систем документообігу є такими:

− синхронізація, підтримка транзакцій, розв’язання конфліктів у разі одночасних змін документів за рахунок локальних механізмів блокування. Підтримки блокування декількох документів, розміщених на різних серверах, немає. Лише останнім часом з'явилися пропозиції з підтримки колективного редагування з блокуванням спільно використовуваних документів;

− стійкість до відмов, для забезпечення якої системи документів використовують різні методики, зокрема надійність системи зв'язку забезпечує протокол TCP, а кластери, які застосовують для підтримки відмовостійкості, підвищують доступність і збільшують продуктивність web-сервера. Запропоновано і реалізовано кілька аналогічних рішень, більшість з яких полягають у використанні виділеного сервера зовнішнього інтерфейсу, що передає запити на один із серверів кластера;

− захист, керування доступом, автентифікація, яка ґрунтується на використанні сертифікатів автентифікації, при цьому особливу увагу приділяють перевірці дійсності сертифікатів. Для того щоб вирішити, чи можна довіряти відкритому ключу, що міститься в сертифікаті, у системі підтримується специфічна модель довіри. Звичайна довіра встановлюється відповідно до ієрархічної схеми іменування, яка частково конфігурується у процесі налаштування сервера. З допомогою перехресних сертифікатів дві різні системи можуть висловити одна одній взаємну довіру, що дозволяє їх клієнтам задіяти сервери кожної з цих систем.

6.5.2. Вимоги до корпоративних інформаційних систем

Усі названі вимоги до корпоративних інформаційних систем є загальними вимогами, але крім них системи мають підтримувати порядок роботи з документами, визначений підприємством. Найпростіший випадок документообігу зображено на рис. 6.6, коли працівник готує документ, відправляє його на перевірку та підтвердження, а далі розсилає всім необхідним отримувачам документа.

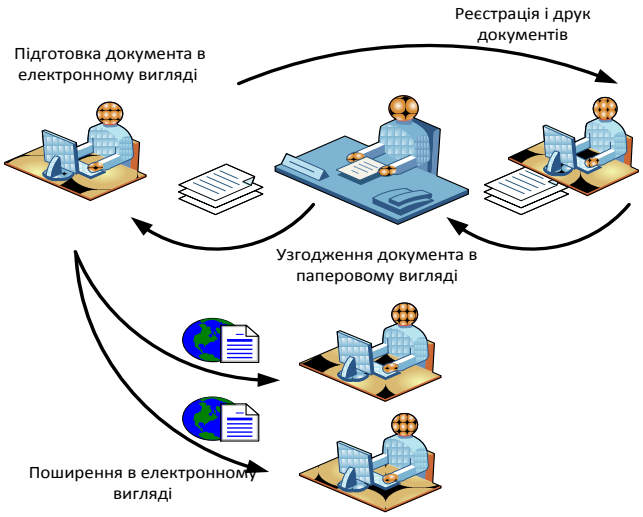


Рис. 6.6. Найпростіший документообіг на підприємстві

Крім такого обігу документів, для забезпечення повної функціональності система має містити інтерфейс введення даних, редагування, збереження версій документів з можливістю переглянути документ на будь-якій стадії його розробки, а також повернутися до однієї з попередніх версій. Проста схема роботи з документами в офісі вимагає засобів розсилання повідомлень, тому застосовують розсилання повідомлень про статус документа, необхідність перевірки та підпису. Пошук документів найпростіше здійснювати за його порядковим номером, для формування якого використовують спеціальні алгоритм його присвоєння, але пошукова система має

виконувати запити за різними даними, а саме за автором документа, назвою, номером, останньою датою редагування. Здебільшого документи необхідно зберігати тривалий час, випадково не знищивши чи пошкодивши їх. Захист від відмов та пошкоджень баз даних є основною вимогою корпоративних систем документів.

6.6. Наявні системи документів

Нині існує дуже багато систем створення, редагування, зберігання документів. Розглянемо принцип дії деяких із них.

6.6.1. Microsoft Exchange Server

Microsoft Exchange Server – надійна система обміну повідомленнями із вбудованими засобами захисту від небажаної пошти і вірусів. За допомогою Exchange Server користувачі організації отримують доступ до електронної, голосової пошти, календарів і контактів з використанням широкого спектра пристроїв і з будь-якого місцезнаходження.

Цей сервер характеризується підвищеною безпекою і надійністю, оскільки дозволяє забезпечити доступ до корпоративної інформації співробітникам підприємства фактично з будь-якого місця і в будь-який момент.

Microsoft Exchange Server охоплює п'ять ролей сервера, які можна встановити і настроїти на комп'ютері, на якому він працює, зокрема такі:

1. Клієнтський доступ (Client Access, CA).
2. Граничний транспорт (Edge Transport, ET).
3. Транспортний сервер-концентратор (Hub Transport, HT).
4. Сервер поштових скриньок (Mailbox Server, MB).

5. Єдина система обміну повідомленнями (Unified Messaging, UM).

Роль сервера «Клієнтський доступ» підтримує клієнтське прикладне програмне забезпечення Microsoft Web-клієнт Outlook і Microsoft Exchange ActiveSync. Ця роль підтримує під'єднання до сервера Exchange різних клієнтів. Програмні клієнти, зокрема Microsoft Outlook Express, використовують з'єднання POP3 і IMAP4, а апаратні, наприклад мобільні пристрої, використовують ActiveSync, POP3 чи IMAP4 для зв'язку із сервером Exchange.

Для виконання ролі граничного транспортного сервера застосовують в зоні підприємства автономний сервер, який створюють для зменшення площі атаки, обробки всього поштового потоку, що надходить з Internet, забезпечуючи передачу протоколом SMTP і роботу служб проміжних вузлів Exchange. Додаткові рівні захисту й безпеки повідомлення забезпечують агенти, які запущені на граничному транспортному сервері й виконують операції з повідомленнями під час їх обробки компонентами транспортування повідомлень. Ці агенти підтримують засоби, що забезпечують захист від вірусів і небажаної пошти, застосовують правила транспортування для керування потоком повідомлень.

Роль транспортного сервера-концентратора розгорнута всередині служби каталогів Active Directory, керує всім потоком пошти на підприємстві, застосовує правила транспортування і політики ведення журналу і доставляє повідомлення в поштову скриньку отримувача. Повідомлення, відправлені в Internet, вузловий транспортний сервер передає до ролі сервера граничного транспорту, розгорнутої на периметрі мережі. Повідомлення, перш ніж передаватися серверу вузлового транспорту, обробляє сервер граничного транспорту. Якщо сервера граничного транспорту немає, то можна налаштувати сервер вузлового транспорту для безпосереднього передавання повідомлень. На сервері вузлового транспорту можна також установити і настроїти

агентів граничного транспорту, які забезпечать в організації захист від небажаної пошти і комп'ютерних вірусів.

Роль сервера поштових скриньок забезпечує збереження баз даних поштових скриньок користувачів. Якщо в поштовій системі планується зберігати поштові скриньки користувачів, загальні папки, або більше даних, роль сервера поштових скриньок є обов’язковою. У Exchange Server роль сервера поштових скриньок інтегрована зі службою каталогів Active Directory. Роль сервера поштових скриньок розширює інформаційні можливості співробітників, забезпечуючи покращені функції календаря, керування ресурсами й автономним завантаженням адресних книг.

Роль сервера єдиного обміну повідомленнями поєднує голосові повідомлення, факс і електронну пошту в одній папці вхідних повідомлень, до якої можна отримати доступ з телефону і комп'ютера. Єдина система обміну повідомленнями поєднує Exchange Server з телефонною мережею організації та надає функції єдиної системи обміну повідомленнями для Exchange Server.

У процесі взаємодії Exchange з декількома клієнтами безпеку забезпечують міжмережні екрани разом із сервером Microsoft ISA Server, який відіграє роль шлюзу і додатково захищає Exchange й інші компоненти на серверній стороні.

Взаємодія з Outlook. Під час взаємодії Outlook з Exchange забезпечується усталена робота у разі ненадійних, низькошвидкісних або неякісних з'єднань.

Outlook підтримує такі основні фукції:

- режим хешування даних Exchange, який дозволяє одержувати доступ до повідомлень, навіть якщо немає з'єднання із сервером Exchange;
- з'єднання без використання VPN на основі протоколу RPC4.

Програмний засіб Outlook Web Access, який функціонує як сервіс, є ефективним і безпечним та підтримує такі функції:

- засоби перевірки правопису;
- підтримку списку задач;
- блокування HTML-коду і вкладень, щоб уникнути відправлення повідомлення про те, що користувач відкрив повідомлення, і подальшого одержання спаму;
- автоматичне завершення сеансу зв'язку (якщо користувач забув завершити сеанс зв'язку, після закінчення визначеного періоду простою сеанс завершується автоматично);
- підтримка S/MIME для Outlook Web Access дозволяє використовувати цифрові підписи і шифрувати повідомлення електронної пошти.

Мобільні пристрої на основі Windows, зокрема карманні персональні комп'ютери (КПК), постачають із вбудованим програмним забезпеченням Microsoft ActiveSync і Pocket Outlook, що дозволяє синхронізувати електронну пошту, календар і списки контактів безпосередньо з Exchange.

Користувачі Exchange за допомогою Outlook Mobile Access можуть одержувати доступ до поштових скриньок з мобільних пристроїв, оснащених браузерами з підтримкою HTML. Гнучкі засоби доступу до даних і нові технології безупинного доступу дозволяють користувачам підвищити продуктивність праці й самостійно визначати час і спосіб взаємодії.

6.6.2. Windows SharePoint Services

SharePoint Server – це серверна платформа, призначена для забезпечення спільної роботи, надання засобів керування змістом,

упровадження бізнес - процесів і надання доступу до інформації, важливої для організаційних цілей та процесів.

За допомогою шаблонів вузлів й інших засобів SharePoint Server можна швидко та ефективно створювати вузли, які підтримують публікацію визначеного контенту, керування контентом і записами. Наприклад, можливе створення вузлів рівня організації, зокрема корпоративних порталів у інтрамережі або Web-вузлів, спеціалізованих вузлів (інформаційних сховищ довільної структури), які дозволяють спільно працювати й обмінюватися даними користувачам як усередині організації, так і за її межами. Крім того, SharePoint Server можна використовувати для здійснення ефективного пошуку людей, документів і даних, для створення бізнесів-процесів на основі форм і участі в них, а також для доступу до великих обсягів бізнес-даних та їх аналізу.

Переваги використання SharePoint Server такі:

1. *Ефективна спільна робота з іншими користувачами на підприємстві.* Ця перевага реалізується засобами ведення календарів для перегляду запланованих подій групи, бібліотек документів - для збереження документів групи або певного підрозділу організації; засобами обговорення виробничих питань, використовуючи блоги, записування і збереження звітів на сторінках, які є керованими користувачами базами знань.

2. *Створення особистих вузлів*, на яких користувачі можуть керувати власними інформаційними ресурсами і надавати доступ до них іншим користувачам, зокрема для централізованого перегляду і керування певними інформаційними ресурсами, задачами тощо.

3. *Пошук людей і даних* у бізнес-орієнтованому прикладному програмному середовищі, тобто можна знаходити дані в корпоративній базі даних або корпоративних системах, зокрема системах керування

взаємовідносинами з клієнтами (Customer Relationship Management, CRM).

4. *Керування документами, записами і Web-умістом*. Наприклад, організація може розробити процес припинення дії документів після визначеного часу.

5. *Розміщення бізнес-форм на основі XML*, інтегрованих із базами даних чи іншим бізнес-орієнтованим прикладним програмамним середовищем. Наприклад, для певного підрозділу корпорації можна розробити форми заяв у Microsoft Office InfoPath і розмістити їх у SharePoint Server, щоб потім користувачі могли заповнювати ці форми безпосередньо у браузері, при цьому введені дані відправлятимуть в базу даних у мережі установи.

Технології SharePoint реалізуються набором служб Microsoft Windows SharePoint Services (WSS), які дозволяють створювати і підтримувати Web-сайти, за допомогою яких члени групи можуть взаємодіяти, обмінюватися документами і спільно працювати над проектами.

Microsoft Office дозволяє ефективніше використовувати WSS, зокрема звертатися до сайту так, начебто він є частиною локальної системи, зберігати файли в бібліотеки, редагувати документи в кожній з програм Office, переміщати будь-яку інформацію на сайт тощо. За допомогою Web-сервісів продукти і технології SharePoint використовують інформацію фактично з будь-якого корпоративного інформаційного середовища.

WSS суттєво полегшує операції резервного копіювання і відновлення, якщо відповідні дані розподілені багатьма різноманітними системами збереження. Всі документи, списки і конфігураційна інформація розміщуються в базах під керуванням SQL Server, що спрощує керування операціями копіювання і відновлення, а також

забезпечує високу масштабованість і підтримку персоналізованого подання Web-порталу для користувачів.

Microsoft WSS є технологічною платформою для SharePoint Portal Server - серверного продукту з додатковою функціональністю для створення надійного, масштабованого, простого у використанні й керуванні порталу, для підтримки колективної роботи SharePoint Portal Server і Office SharePoint Server (MOSS) забезпечують високий рівень спільної роботи над документами. Функції керування документами, вбудовані в Office SharePoint Server, можуть бути задіяні в будь-яких рішеннях на основі продуктів і технологій SharePoint. Так, файловий сервер є не звичайним сховищем документів, а повноцінним корпоративним порталом з підтримкою пошуку, категоризації, розгляду документів і одержання сповіщень про зміни в них.

Використання SharePoint Server має такі переваги:

1. Підвищення ефективності роботи груп:

− спрощення і прискорення обміну даними і результатами роботи груп за рахунок співробітництва в режимі реального часу;

− економія засобів за рахунок спрощення взаємодії всередині групи;

− більш просте керування процесами за допомогою надійного контролю версій;

− спрощення керування проектом за рахунок застосування вдосконалених засобів і функцій.

Підвищення ефективності процесу в межах організації:

− раціональне адміністрування вузлів, сховищ і систем безпеки;

− зручність колективної роботи в межах груп за рахунок інтеграції з Microsoft Outlook і Office;

− прості й багатофункціональні засоби настроювання;

− покращені засоби узагальнення даних за рахунок використання XML і Microsoft InfoPath;

− підвищення рівня безпеки і зниження ризиків за рахунок автоматизації захисних заходів;

− більш висока безпека роботи в Internet.

Платформа для розвитку організації:

− покращене використання і доступність даних із різних систем;

− застосування Microsoft Office System як ефективного набору засобів для колективної роботи;

− розвиток архітектури системи за рахунок використання стандартних API- інтерфейсів;

− переваги поетапного розгортання;

SharePoint Portal Server порівняно з Office SharePoint Server надає такі додаткові можливості:

− створення, розгортання і керування порталами для невеликих груп, підрозділів великих організацій чи цілих підприємств із сотнями тисяч користувачів, десятками тисяч сайтів і мільйонами документів;

− пошук будь-якої інформації в документах, які має організація, незалежно від їх місцезнаходження – на файлових серверах, Web-сайтах, в інших системах збереження документів чи у корпоративних прикладних програмах;

− підтримка адаптованих структур для створення, організації та пошуку всіх джерел інформації в межах підприємства, зокрема на сайтах відділів і підрозділів, Web-сайтах проектів, над якими працюють групи, персональних Web-сайтах тощо;

− персоналізована категоризація і поширення відомостей та іншої корпоративної інформації з порталу на основі профілів користувачів та аудиторій, визначених даною організацією.

Для якісної підтримки великої кількості сайтів Office SharePoint Server містить стандартні засоби для створення і керування сайтами, тоді як SharePoint Portal Server - засоби розгортання й адміністрування сайтів для великих підприємств, зокрема такі:

- каталог сайтів;
- рішення з інтеграції корпоративних рішень;
- карти сайтів, які динамічно створюються;
- засоби керування великомасштабною топологією серверів;
- можливість спільної роботи множини серверів індексації та пошуку.

У Microsoft WSS реалізована функція персоналізації, яка забезпечує підтримку аудиторії – динамічної групи користувачів з однією або декількома загальними властивостями (бізнес-функцією, відділом, групою користувачів). Приналежність користувача до певної аудиторії визначає можливість його доступу до Web-компонент, фільтри інформації.

Microsoft WSS підтримує такі функції:

- спільну роботу над документами;
- обмін інформацією на рівні груп і проектів;
- керування віртуальними групами;
- взаємодію між індивідуальними особами;
- інтегровані засоби визначення присутності (з Live Communications Server);
- списки і бібліотеки документів;
- контроль версій документів, процедуру взяття файлів на редагування та їх повернення;
- сповіщення про зміни;
- установку додаткових Web-компонентів;
- загальні календарі й засоби підтримки дискусій;
- створення Web-сайтів і керування ними користувачами;
- інтеграцію з Microsoft Office;
- підвищення продуктивності праці окремих осіб і груп;
- повторне використання наявної інформації у процесі створення нових документів;

- розмежування доступу до інформації на основі ролей;
- продуктивне середовище для колективної роботи;
- спрощення використання, розгортання і настроювання.

SharePoint Portal Server містить усі можливості Microsoft WSS, однак додатково забезпечує такі функції:

- структуризацію сайтів у межах усього підприємства;
- створення сайтів SharePoint із застосуванням каталогу сайтів (Site Directory);
- централізоване адміністрування всіх порталів і сайтів груп в організації;
- єдиний вхід у корпоративне прикладне програмне забезпечення;
- контекстний пошук будь-яких даних та інформації в межах усієї організації;
- затвердження документів;
- підвищення ефективності роботи організації;
- більш ефективні засоби пошуку і відбору інформації з будь-якого джерела в межах організації або поза ними;
- суттєве підвищення продуктивності праці завдяки розширеним можливостям повторного використання наявної інформації під час створення нових документів і завдяки більш ефективній взаємодії груп.

Елементи архітектури Windows SharePoint Services. *Вузли* – це обчислювальні елементи структури мережі для розміщення загальної інформації, яка спрощує обмін даними, перевірку обговорень і керування версіями документів. Вузол може містити підвузли, які утворюють ієрархічну структуру на web-серверах, аналогічну дереву папок у файловій системі. Крім зберігання файлів, вузли забезпечують новий рівень обробки даних, надаючи середовище для спільної роботи над документами, задачами, подіями та іншими типами інформації. Такий підхід значно підвищує працездатність як окремих користувачів, так і робочих груп. Типовий вузол може містити як загальні дані

(бібліотеки документів, контакти, календарі, списки задач, обговорення), так і корисні інструменти, призначені для редагування і відображення інформації.

Користувачі можуть переглядати зміст вузла SharePoint, здійснювати пошук контактів, зв'язуватися з ними за допомогою електронної пошти, а також одержувати повідомлення у разі зміни наявної інформації та додавання нової. Вміст і макет вузла можна налаштувати таким чином, щоб подавати визначені відомості вказаним користувачам за заданими темами.

Для спільної роботи над певною задачею або документом проект SharePoint Services створює спеціальне середовище, яке називають робочою областю для документів, створеною за допомогою Office або у вікні браузера.

У багатьох організаціях спільно розробляють документи за допомогою загальних мережних папок і електронної пошти. Відповідно до цієї методики, створивши документ, користувачі відправляють його на рецензування колегам, вклавши в повідомлення електронної пошти. В Office можна автоматично створити робочу область для документа і загальне вкладення для його пересилання. Загальне вкладення – це документ, який зберігається у вузлі SharePoint і є пов'язаним з повідомленням електронної пошти; у процесі створення якого SharePoint зберігає документ, після чого можна настроїти робочу область для відстеження версій документа. Активізувавши відстеження версій, користувачі можуть переглядати попередні версії документа, а у разі потреби скасовувати зміни.

Права користувачів у Windows SharePoint Services. У WSS доступ до вузлів контролює система членства, яка ґрунтується на ролях і реалізується через групи вузла. Кожен користувач вузла SharePoint належить принаймні до однієї групи прямим або непрямим чином.

Для кожної групи визначено набір правил, що описує дії, які члени групи можуть виконувати у вузлі SharePoint. За замовчуванням у WSS є такі п'ять груп користувачів вузла:

- гість – ця група не надає ніяких прав;
- читач – члени цієї групи можуть переглядати зміст вузла;
- кореспондент – доповнюючи права читача, члени цієї групи мають право додавати, змінювати і видаляти елементи вузла, наприклад документи;
- web-дизайнер – крім прав учасника, члени цієї групи мають право модифікувати сторінки вузла;
- адміністратор – члени цієї групи мають повний доступ до вузла.

Типовий web-вузол WSS надає користувачам інфраструктуру для взаємодії, обміну інформацією, збереження загальних даних і спільної роботи. Розрізняють такі типи web-вузлів: вузол групи, порожній вузол, робоча область для документів. Інфраструктура вузла групи містить такі компоненти:

- бібліотеки, тобто набори документів, рисунків, форм, які спільно використовують члени робочої групи, зокрема вбудовану бібліотеку документів, яку називають *«Загальні документи»*;
- списки, які дозволяють членам групи працювати зі структурованими, табличними даними, які зберігаються у web-вузлі, переважно вузол групи має такі п'ять вбудованих списків, як *«Оголошення», «Контакти», «Події», «Зв'язки»* і *«Задачі»*. Крім них, до вузла можна додати й інші списки, які містяться у WSS, а також створити власні списки;
- дошки обговорення – надають місце (форум), де члени групи можуть розміщувати свої зауваження і відповідати на зауваження колег. За замовчуванням типовий вузол групи постачається із вбудованою дошкою обговорення *«Загальні обговорення»* та дозволяє створювати власні дошки обговорення;

− огляди, які забезпечують проведення опитувань користувачів вузла. Вузли не мають вбудованих оглядів, але їх можна створити.

Переміщення за ієрархією вузла. Вузол може мати дочірні вузли (підвузли), ієрархія яких нагадує ієрархію папок у файловій системі. Вузли, які не мають батьківських, називають вузлами верхнього рівня, до яких також належать підвузли, кожний з яких може мати власні підвузли, і так далі. Ієрархічну структуру вузла називають родиною вузлів, яку зображено на рис. 6.7.

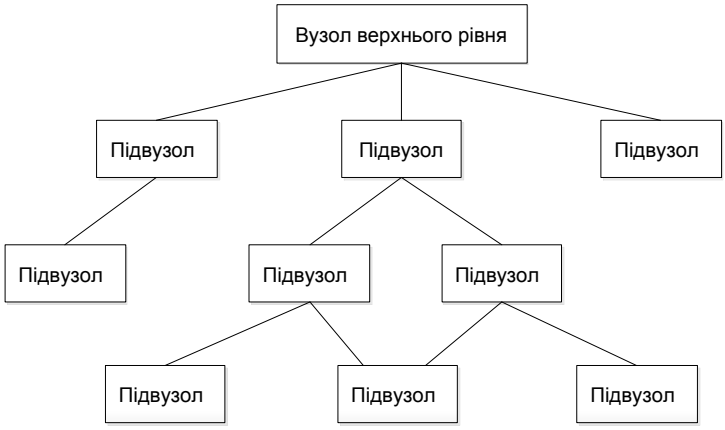


Рис. 6.7. Родина вузлів

З появою четвертого покоління SharePoint змінено імена WSS та MOSS на SharePoint Foundation, основну функціональність якого значно удосконалено порівняно з WSS, та SharePoint Server, який складається з двох функціональних рівнів поверх SharePoint Foundation, відповідно.

SharePoint Foundation має такі часто використовувані функції та характеристики:

− web-орієнтоване прикладне програмне забезпечення, яке працює поверх Internet Information Services (IIS);

− використовує 64-розрядну операційну систему Windows Server;

− усі дані та інформація зберігаються в одній або більше базах даних Microsoft SQL Server;

− на екран інформація виводиться з застосуванням файлів web-сторінок, які зазвичай містять одну або більше web-частин;

− наявні зручні функції керування документами, призначеними для користувача, зокрема історія версій, метадані та інтеграція з Microsoft Office;

− є багато типів списків, які можна використовувати для зберігання різних типів інформації, таких як документи, контакти, календарі;

− дозволяє створювати workflow-рішення, відправляти електронну пошту користувачеві після зміни вказаного документа або встановлення певного значення поля списку;

− дозволяє створювати базові ефективні рішення для Intranet з власними вбудованими функціями керування web-контентом;

− підтримує співпрацю над даними проектів, організацію зустрічей, соціальних заходів, блоги тощо.

Однак SharePoint Foundation не виконує таких важливих функцій:

− обмежена функціональність розширеного пошуку, через що користувачі можуть виконувати пошук тільки на поточному сайті або на сайтах нижчих за рівнем;

− розширених функцій керування web-контентом, зокрема керування публікаціями, цільовою інформацією, багатомовної підтримки;

− розширених функцій керування документами (глобальної ідентифікації (identification, ID) документа, наборів документів, політики документів);

− керування юридичними записами та іншими важливими документами;

− підтримки відображення форм InfoPath у web-браузері, електронних таблиць MS Excel та діаграм MS Visio як web-частин;

− підтримки ключових показників ефективності (KPIs).

SharePoint Server використовує такі самі типи web-сайтів і функцій, що й SharePoint Foundation, але додатково забезпечує такі функції:

− використання функціональності глобального пошуку, який грунтується на контенті або властивостях метаданих, для пошуку будь-якої інформації, незалежно від типу і розташування;

− використання «соціального пошуку» для знаходження людей, який грунтується на їх типових діях та інтересах;

− цільове виведення інформації одній або більше групам користувачів;

− імпортування інформації про користувачів з адміністративного домену (Administrative domain, AD) та її доступність для пошуку;

− використання розширених функції керування контентом для загальнодоступних сайтів або сайтів Intranet-порталу;

− глобально унікальні ID документів і наборів документів;

− відображення і використання форм InfoPath web-клієнтом за допомогою Forms Service;

− відображення електронних таблиць Microsoft Excel і діаграм у web-частині, використовуючи Excel Services;

− відображення схем Microsoft Visio безпосередньо на web-сторінці засобами web-частин Visio;

− пошук, відображення і редагування контенту в зовнішніх базах даних, таких як SAP, Oracle, і Microsoft SQL, використовуючи Business Connectivity Service;

− надання кожному користувачеві SharePoint особистого web-сайту для приватного і загальнодоступного використання;

− створення інструментальних панелей з показниками оцінювання і ключовими показниками ефективності.

6.6.3. Допоміжні програмні засоби для роботи з розподіленими системами документів

Microsoft Office InfoPath – гнучкий та ефективний засіб збирання інформації в динамічні форми, її поширення і повторного використання в межах групи чи підприємства. InfoPath сприяє успішному веденню бізнесу, розширюючи можливості колективної роботи і поліпшуючи процес прийняття рішень. Інформацію, зібрану за допомогою InfoPath, можна інтегрувати з web-сервісами і різноманітними бізнес-процесами, тому що InfoPath підтримує будь-які XML-схеми користувачів. InfoPath може стати частиною як формалізованих, так і неформалізованих бізнес – процесів у сучасних організаціях.

InfoPath забезпечує такі переваги:

− полегшує збирання потрібних даних, паралельно перевіряючи їх на допустимість, виводячи на екран підказки і форматуючи зібрану інформацію за заданими правилами;

− класифікує зібрану інформацію, дозволяючи додавати розділи в результуючих формах;

− працює із формами як в онлайновому, так і в автономному режимах, що дає можливість керувати даними в будь-якому місці й у будь-який час;

− надає зручне середовище й інструментарій Microsoft Office, що різко скорочує витрати на навчання;

− поєднання людей, інформації та процесів спрощує повторне використання зібраних даних;

− дозволяє обмінюватися інформацією та повторно використовувати її між різними системами і процесами за рахунок підтримки технології web-сервісів;

− поліпшує умови колективної роботи у групах, тому що може взаємодіяти з Office SharePoint Server.

Крім того, InfoPath спрощує розробку і розгортання динамічних форм у межах усього підприємства. Ці форми можна публікувати в загальному каталозі мережі, на web-сервері, у бібліотеці форм Windows SharePoint Services чи пересилати електронною поштою.

InfoPath - це прикладне програмне забезпечення, за допомогою якого можливо прямо з робочого столу отримувати доступ до віддалених даних, а також інтелектуальний клієнтський доступ до web-сервісів без програмування.

InfoPath працює у двох основних режимах: конструювання форми, тобто редагування готового шаблону, та її заповнення, тобто введення даних у поля, які можуть бути текстовими, списками, пропорціями тощо. Після завершення введення форму можна зберегти або опублікувати (ця функція забезпечується за допомогою майстра публікування форм).

Для створення нової форми, не користуючись шаблоном є декілька варіантів дій: сконструювати форму «з нуля», додаючи й описуючи різні її компоненти; використовувати вже наявний XML-документ чи XML-схему; створити за допомогою під'єднання зовнішнього джерела даних (Access, SQL Server або web-сервісів); імпортувати форму з web-сайту; використати механізм внутрішнього програмування скриптовими мовами для розробки більш функціонально насичених рішень.

Технології в InfoPath. В основу програмних засобів InfoPath, за допомогою яких створюють динамічні форми, покладено широке використання різних XML-технологій: XML, XPath, XSD, XSLT, XHTML, CSS, DOM, XML DSig, SOAP, WSDL, UDDI. Загальну логіку формування динамічних форм подано на рис. 6.8. Вихідні дані отримують із XML-файлів або SOAP-повідомлень (web-сервісів), з яких за допомогою XML-схеми створюється внутрішнє дерево даних документна об’єктна модель (Document object model, DOM). Зовнішній

вигляд документа формують з використанням таблиць стилів XSLT. Однак слід зауважити, що всі ці програмні технології містяться всередині прикладної програми – користувач керує ними через відповідний візуальний інтерфейс.

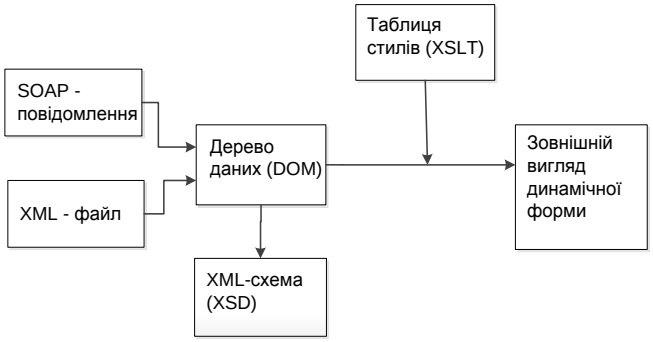


Рис. 6.8. Механізм формування динамічної форми

Для створення прикладного програмного забезпечення на основі Microsoft Office InfoPath можна використовувати інструментальні засоби Software Development Kit (SDK), які дозволяють застосовувати різні підходи для більш гнучкого настроювання і під'єднання функціонала InfoPath, зокрема його можна інтегрувати з Access, Word, Microsoft Windows SharePoint Services, Microsoft BizTalk Server, Microsoft SQL Server, XML Web Services, Component Object Model (COM-об'єктами) і Active Data Objects (ADO). Так, застосовуючи ADO, можна задіяти InfoPath як «збирача» інформації з найрізноманітніших зовнішніх джерел даних, а через BizTalk Server - інтегрувати його в бізнес-процеси підприємства. Для розширення функціональних можливостей InfoPath використовують додатковий інструментарій InfoPath Toolkit for Visual Studio .NET, який дозволяє створювати проекти InfoPath у середовищі Visual Studio за допомогою мов C# або Visual Basic .NET і .NET Framework.

Microsoft ISA Server реалізує функціональні можливості екрана, засобу керування приватними віртуальними мережами, служби web-кешування, дозволяє підвищити безпеку і продуктивність

корпоративної інформаційної мережі, а також знизити експлуатаційні витрати. ISA Server має такі особливості:

− наявні засоби захисту, що реалізують динамічну фільтрацію пакетів і каналів, алгоритм роботи якої відкриває доступ пакетів даних до захищених зон мережі, у разі потреби служба динамічної фільтрації відкриває порти, а по завершенні сеансу зв'язку – закриває;

− простота використання за рахунок підтримки багаторівневої архітектури, уніфікації керування віртуальною приватною мережею (Virtual Private Network, VPN), зрозумілих шаблонів, удосконалених засобів видалення неполадок, можливості експортування конфігурації у форматах XML, моніторингу активних з'єднань у режимі реального часу;

− швидке і надійне одержання доступу до віртуальної приватної мережі за рахунок вбудованої підтримки тунельного режиму комплектом протоколів для передавання інформації у віртуальних приватних мережах (IP Security, IPSec) для VPN-підключень, швидке web-хешування і високопродуктивний пакетний фільтр.

Microsoft Office Live Communications Server забезпечує високі надійність, керованість, захист конфіденційної інформації та загальну безпеку. Цей продукт надає для бізнесу ефективний контроль за конфіденційною інформацією у межах усієї організації.

Функціональність миттєвого обміну повідомленнями і визначення присутності користувачів (можливості з'ясувати, чи доступний колега на одному або декількох пристроях) в Live Communications Server є частиною масштабованого корпоративного рішення, що забезпечує високий рівень безпеки й безшовну інтеграцію з іншими продуктами Microsoft, а також надає розширювану платформу розробки на засадах промислових стандартів, забезпечує передачу текстових повідомлень у режимі реального часу IP-мережею, зокрема Internet або корпоративною мережею. Live Communications Server забезпечує

спільне використання прикладних програм і колективну роботу над даними між одноранговими (рівноправними) хостами (вузлами) у мережі, проведення аудіо- і відеоконференцій, кардинально прискорюючи всі операції. Public IM Connectivity – це можливість з'єднання наявної бази користувачів Live Communications Server із загальнодоступними службами миттєвого обміну повідомленнями. Користувачі загальнодоступних мереж і Live Communications Server можуть взаємодіяти у режимі реального часу так само, як із колегами свого підприємства.

Live Communications Server інтегрується легко з Microsoft Office і системами Windows Server, інтегрується з наявними бізнес-процесами й IT-інфраструктурою.

Основні переваги Live Communications Server такі:

− економія витрат і більш висока продуктивність порівняно з аналогічними системами колективної роботи за рахунок миттєвого обміну повідомленнями й визначення присутності користувачів, що сприяє скороченню витрат і прискореному прийняттю більш обґрунтованих рішень;

− інтеграція з Microsoft Office Communicator надає можливість використовувати прості засоби пошуку контактів за допомогою служби адресної книги Live Communications Server. Співробітники можуть шукати колег за глобальним для корпорації списком адрес (Generic Array Logic, GAL), а також на основі локальних баз даних адрес, що зберігаються на особистих комп'ютерах.

− інтеграція з Microsoft Office Outlook і Microsoft Exchange Server дозволяє одержувати інформацію «вільний/зайнятий» за будь-якими контактами прямо з розкладу графіка робіт, а також виводити власні повідомлення «поза офісом» безпосередньо в Office Communicator;

− розширені засоби визначення присутності, у тому числі можливість створювати «нотатки користувача», дають більш

інформативні відомості іншим контактам про те, як найкраще зв'язатися, та функціонують незалежно від того, під’єднаний користувач чи працює в автономному режимі;

− за наявності належної інфраструктури шлюзу з офісними телефонними системами (Private Branch Exchange. PBX) або комутованими загальнодоступними телефонними мережами (Public Switched Telephone Networks, PSTN) Office Communicator забезпечує інтеграцію з ними, що дозволяє керувати офісним телефоном безпосередньо з комп'ютера для ініціації телефонних викликів і навіть для перенапрямлення вхідних викликів, якщо співробітника немає на робочому місці;

− конференц-зв'язок із партнерами можна організувати прямо з Office Communicator, що істотно полегшує взаємодію між співробітниками інформаційних відділів.

За наявності Live Communications Server і партнерських рішень з інтеграції з телефонними мережами Office Communicator підтримує деякі варіанти офісного зв'язку, зокрема керування викликами, перехоплення дзвінків, їх пересилання, а також сеанси Microsoft Office Live Meeting.

6.7. Висновки

1. Розподілені системи документів, особливо WWW, є найпопулярнішим серед кінцевих користувачів мережним прикладним програмним середовищем.

Розрізняють різні способи моделювання документів, найважливіший з яких – спосіб скріплення документів один з одним. Модель гіпертексту, яку підтримують як Web, так і Lotus Notes, суттєво сприяла широкому впровадженню розподілених систем документів завдяки тому, що в ній легко можна активізувати посилання на інший

документ, результатом чого є отримання документа, на який вказує посилання, і виведення його на екран користувача.

Документи зазвичай зберігають на серверах, які надають клієнтам доступ до них. Якщо посилання на документи можуть перетинати межі між серверами, тобто за наявності глобальної системи посилань на документи, порівняно просто організувати глобальний розподіл документів. У такому разі посилання містить ім'я сервера, на якому зберігається документ, і клієнта, котрий бажає отримати документ та напряму звертається до цього сервера.

Важливим аспектом архітектури розподілених систем документів є механізми доступу до документів. Від того, який механізм доступу використовує певна система, залежить рівень її масштабованості. Документи зазвичай редагує їх єдиний власник або невелика група користувачів, а читати - достатньо велика кількість користувачів, заходи щодо забезпечення захисту полягають в організації захищеного каналу і подальшому контролі доступу.

Серед розподілених систем документів можна виокремити такі групи систем, які:

- забезпечують збереження та спільне використання документів;
- організують систему пошуку документа за номером, автором, назвою та іншими даними;
- застосовують системи захисту й організації доступу з різними правами;
- забезпечують контроль за поштою, повідомленнями, телефонними дзвінками, викликами;
- утворюють портали, як середовище для створення сайтів, блогів, пошукових баз даних.

Серед допоміжних програмних засобів для роботи з розподіленими системами документів можна назвати такі групи систем, які:

- призначені для розробки форм введення даних;

− застосовують для аналізу відвідуваних ресурсів, обліку трафіку, а також захисту від атак з мережі Інтернет;

− мають можливість обміну миттєвими повідомленнями і функції визначення присутності, й є компонентом масштабованого корпоративного середовища.

6.8. Запитання для самоконтролю

1. Що називають розподіленою системою документів? Наведіть приклади.
2. Яким чином відбувається посилання на документи в WWW?
3. Які типи документів використовуються в WWW?
4. Опишіть загальну структуру Web-систем.
5. Як клієнт взаємодіє з web-сервером?
6. Що являє собою документальна модель WWW?
7. Опишіть архітектуру клієнта і сервера в Web.
8. Що таке CGI та як його застосовують у Web?
9. Чим відрізняються аплети від сервлетів?
10. Основні відмінності Lotus Notes від Web.
11. Порівняйте WWW і Lotus Notes між собою за основними принципами побудови, зв'язком, процесами, призначенням імен, синхронізацією, кешуванням і реплікацією, відмовостійкістю, захистом.
12. Яким вимогам має відповідати розподілена система документів на підприємствах?
13. Які типи систем керування документами виокремлюють?
14. Яка основна архітектура використовується в розподілених системах документів?
15. Які системи розподілених документів на сьогодні існують?

16. Що таке Windows SharePoint Services? Наведіть приклад використання.

17. Які права доступу прописано в Windows SharePoint Services?

18. Що дозволено користувачам з правами кореспондента?

19. Що таке InfoPath? Де використовують цей інструментарій?

20. Яку технологію покладено в основу InfoPath?

21. Охарактеризуйте кроки у процесі створення форми засобами InfoPath в режимі конструювання «з нуля».

22. Що таке XML? Які правила створення XML-документа?

23. Назвіть переваги використання Exchange Server.

24. Які функції наявні в ISA Server?

25. Які послуги забезпечує Live Communications Server?

26. Які переваги використання SharePoint Portal Services?

27. Що таке вузол? Що на ньому зберігається?

28. Яким чином Exchange Server взаємодіє з Microsoft Outlook?

29. Назвіть ролі, які дозволяє виконати Microsoft Exchange Server.

30. У чому полягає завдання Microsoft Exchange Server, як сервера поштових скриньок?

31. Назвіть вимоги до розподілених систем документів. Які додаткові вимоги висуваються до корпоративних систем?

32. Наведіть приклади допоміжних програмних засобів для роботи з розподіленими системами документів. Які функції вони виконують?

33. Наведіть приклади сучасних систем електронного керування документами на підприємстві.

7. РОЗПОДІЛЕНІ СИСТЕМИ УЗГОДЖЕННЯ

7.1. Моделі узгодження

Основним підходом, який використовується в системах узгодження, є відокремлення власне обчислювальних процесів від механізмів їх узгодження. Якщо розглядати розподілену систему як набір процесів (у тому числі, багатопотокових), то стане зрозумілим, що обчислювальну частину розподіленої системи становить група процесів, кожен з яких здійснює конкретні обчислювальні операції, причому ці операції можуть виконуватися незалежно від інших процесів.

Якщо процеси володіють зв'язністю посилань і часу, то узгодження здійснюється безпосередньо, тому його називають прямим узгодженням (direct coordination). Зв'язність посилань зазвичай має вигляд явної ідентифікації співбесідника у процесі взаємодії. Так, один процес може взаємодіяти з іншим лише за умови, що він знає ідентифікатор процесу, з яким хоче обмінятися інформацією. Тимчасова зв'язність означає, що обидва процеси, які взаємодіють, активні одночасно.

Якщо процеси не зв'язані за часом, але зв'язані за посиланнями, то таке узгодження називають узгодженням через поштову скриньку (mailbox coordination). За таких умов для взаємодії зовсім не потрібно, щоб два процеси виконувалися одночасно, замість цього взаємодія відбувається за допомогою посилання повідомлень до поштової скриньки, яку можливо використовують спільно.

Комбінація зв'язності за часом і незв'язності за посиланнями формує тип моделей узгодження, який називають «узгодження під час зустрічі» (meeting-oriented coordination). У незв'язній за посиланнями системі процеси не мають повної інформації один про одного, тобто,

коли процесу потрібно погодити свою діяльність з іншими процесами, він не може звернутися до них безпосередньо. Натомість використовується модель зустрічі, на якій збираються процеси, щоб скоординувати свою діяльність, яка передбачає, що процеси, які беруть участь у зустрічі, виконуються одночасно.

Системи узгодження типу «узгодження під час зустрічі» часто реалізуються на основі подій, схожих із тими, які використовуються в розподілених системах об'єктів. Іншим є механізм реалізації «зустрічей», який здійснюють системи публікації/підписки (publish/subscribe systems), у яких одні процеси можуть підписуватися на повідомлення, які містять інформацію з певної теми, а інші — публікувати (тобто створювати) такі повідомлення. Більшість систем публікації/підписки вимагають, щоб взаємодійні процеси були активними одночасно, отже, вони зв'язані за часом. Проте процеси, що взаємодіють таким чином, можуть залишатися невідомими один одному.

Найбільш відомий варіант узгодження – це поєднання не зв'язаних за часом і за посиланнями процесів, в основі якого лежить генеративний зв'язок (generative communication), який вперше було реалізовано у програмній системі Linda. Відмінна особливість генеративного зв'язку полягає в тому, що набір незалежних процесів може використовувати простір даних, який розділяється між ними (колективний), організовуваний за допомогою кортежів та який підлягає зберіганню. Кортежі – це іменовані записи, що містять декілька (але, можливо, і жодного) типізованих полів. Процес може поміщати у колективний простір даних записи будь-яких типів, тобто генерувати зв’язані записи. На відміну від електронних дошок оголошень, у цьому разі немає потреби точно задавати структуру кортежів. Для розподілення кортежів відповідно до інформації, яка в них міститься, достатньо їх імен.

Особливість колективних просторів імен полягає в тому, що вони реалізують механізми асоціативного пошуку кортежів, тобто коли процесу потрібно витягнути кортеж із простору даних, йому достатньо визначити значення полів, які його цікавлять. Будь-який кортеж, що задовольняє опису, може бути отриманим з простору даних і переданим процесу. Якщо нічого знайдено не буде, то процес може заблокуватися до надходження чергового кортежу.

7.2. Система TIB/Rendezvous

Систему TIB/Rendezvous спочатку було описано в поняттях інформаційної шини (information bus), мінімальної комунікаційної системи групи процесів, яка ґрунтується на таких принципах [2].

По-перше, ступінь залежності комунікаційної системи від програмного забезпечення ядра дуже низький, тому що в програмному забезпеченні ядра зовсім не використовують складну семантику впорядкування повідомлень, оскільки передбачено, що ці питання вирішуються на прикладному рівні.

По-друге, повідомлення описують себе самі. На практиці це означає, що прикладна програма може перевірити вхідне повідомлення, щоб визначити, якою є його структура і які дані воно містить. Зауважимо, що на відміну від цього правила в більшості комунікаційних систем передбачено, що процес вже обізнаний про формат вхідних повідомлень і йому залишається лише правільно інтерпретувати їх зміст.

По-третє, процеси не мають зв'язності за посиланнями, що спричинено тим, що у разі обслуговування системи, яка працює, не має відбутися її зупинення, а також необхідно спростити додавання нових процесів «на льоту». Ці вимоги простіше виконати в тому разі, якщо

процеси явно не посилаються один на одного, тому незв'язність за посиланням забезпечується використанням адресації за темою.

Модель узгодження. Система TIB/Rendezvous грунтується на моделі узгодження, за якою процеси не володіють зв'язністю за посиланнями, але можуть отримувати її на короткий термін, що нагадує тип моделі «узгодження під час зустрічі». Система має службу, за допомогою якої процеси отримують можливість залишатися не зв’язаними за часом.

Ключовий механізм, що лежить в основі моделі узгодження системи TIB/Rendezvous, — це адресація за темою (subject-based addressing), коли процес, який має надсилати повідомлення, не може визначити точне місце призначення. Замість цього він дає повідомленню назву теми (subject name), після чого відправляє його до комунікаційної системи для пересилання мережею.

Одержувачі, у свою чергу, не з'ясовують, від яких процесів вони мають отримувати повідомлення, натомість вони повідомляють комунікаційну систему, які теми їх цікавлять. Комунікаційна система гарантує, що одержувачеві будуть доставлені лише ті повідомлення, які містять дані з теми, яка цікавить одержувача.

Відправлення повідомлення за методом адресації за темою також називають публікацією (publishing). Щоб отримати повідомлення з певної теми, процес має підписатися на неї. Принципи організації систем публікації/підписки, які використовують метод адресації за темою, показано на рис. 7.1.

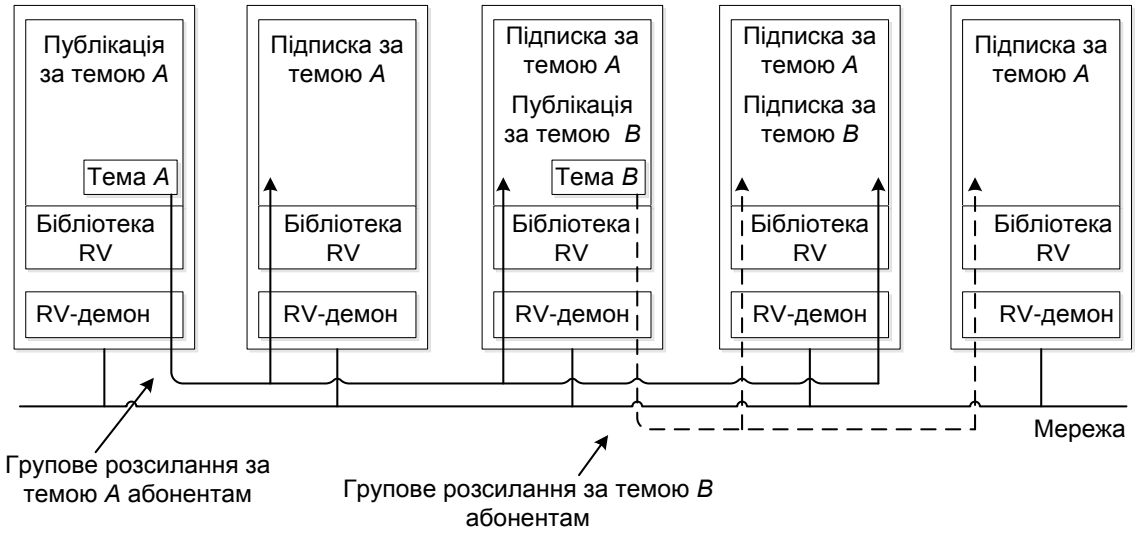


Рис. 7.1. Принципи організації системи публікації/підписки, реалізованої в TIB/Rendezvous

Архітектура системи TIB/Rendezvous порівняно проста та побудована на основі мережі з підтримкою групового розсилання, хоча по можливості допускається використання ефективніших засобів зв'язку. Так, якщо відоме точне місцезнаходження абонента, то зазвичай виконується крізне передавання повідомлень. На кожному вузлі цієї мережі працює демон контактів (rendezvous daemon), який відповідає за те, щоб повідомлення відсилалися і публікувалися відповідно до теми. Коли повідомлення публікується, демон контактів здійснює його групове розсилання всім вузлам засобами базової мережі, зокрема за IP-адресами або за допомогою апаратного широкомовного розсилання.

Процеси, підписуючись на певну тему, передають свою цю інформацію локальному демонові. Демон створює таблицю пар (процес, тема) і під час доставки повідомлення з темою *B* просто проглядає цю таблицю, шукаючи локальних абонентів, пересилаючи її кожному процесу. Якщо на тему *B* на певному вузлі ніхто не підписався, то повідомлення негайно знищується.

Щоб збільшити систему до розміру великих мереж, зокрема до масштабів глобальних або регіональних мереж (Wide Area Network,

WAN), використовують демони, які здійснюють маршрутизацію контактів (rendezvous router daemons). Зазвичай кожна локальна мережа має одного такого демона контактів, котрий зв'язується з аналогічним демоном іншої віддаленої мережі, як показано на рис. 7.2. Демони, які здійснюють маршрутизацію, утворюють наскрізну оверлейну мережу (overlay network), у якій маршрутизатори сполучені між собою попарно з'єднаннями TCP, утворюючи **вторинну мережу,** мережу маршрутизаторів прикладного рівня.

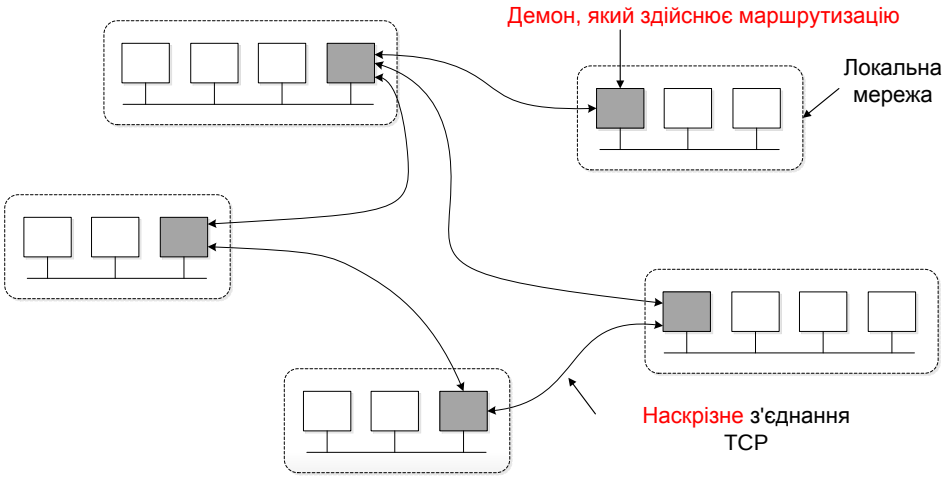


Рис. 7.2. Загальна архітектура глобальної мережі TIB/Rendezvous

Кожний маршрутизатор знає топологію вторинної мережі та визначає дерево групового розсилання для публікації повідомлень в інших мережах. Маршрутизатори розсилають лише ті повідомлення, які публікуються в їх локальній мережі, а повідомлення з інших мереж пересилаються деревом групового розсилання тій мережі, з якої вони спочатку надходили.

7.3. Система Jini

Наступним прикладом системи узгодження є система Jini компанії Sun Microsystems. Jini належить до систем узгодження передусім тому, що вона здатна підтримувати генеративний зв'язок за допомогою Linda-подібної служби JavaSpace.

Система Jini – це розподілена система, яка охоплює різні, але взаємопов'язані елементи. Вона жорстко прив'язана до мови програмування Java, хоча багато з її принципів можливо реалізувати й іншими мовами. Важливою частиною системи є модель узгодження генеративного зв'язку.

Система Jini – це набір інтерфейсів прикладного програмування (Application Programming Interface, API) і мережних протоколів, покликаних допомогти побудувати й розгорнути розподілену систему, організовану як федерація сервісів. Сервіси можуть бути будь-якими процесами обробки інформації, що виконуються в мережі, – апаратними пристроями, здатними працювати під програмним керуванням, прикладними програмами тощо. Наприклад, Jini-сумісні принтери можуть пропонувати сервіс друку, Jini-сумісні відеокамери – сервіс знімання і перегляду зображення. Отже, федерація служб - набір сервісів, які є доступними у мережі в даний момент і якими можуть скористатися клієнти для досягнення певної мети.

Принцип, який покладено в основу побудови служб федерації сервісів, полягає в тому, що інфраструктура Jini не має потреби в центрі керування, а орієнтована на час виконання та забезпечує спосіб клієнтам і сервісам знаходити один одного за допомогою служби пошуку, яка зберігає довідник доступних нині сервісів. Після того як клієнт і сервіси знайдуть один одного, клієнт і залучені ним сервіси взаємодіють не використовуючи інфраструктуру Jini. Якщо службі пошуку Jini не вдасться знайти потрібний сервіс, то така розподілена система, побудована за допомогою сервісу пошуку, продовжить свою роботу. Jini містить також мережний протокол, який можуть використовувати клієнти для знаходження служб у разі відсутності служби пошуку.

Jini визначає інфраструктуру, орієнтовану на час виконання, яка забезпечується механізмом, який дозволяє додавати, видаляти,

знаходити й отримувати доступ до сервісів. Така інфраструктура складається з трьох ланок: сервісу пошуку, постачальників сервісів (таких як Jini-сумісні пристрої) та клієнтів. Служба пошуку є найважливішим механізмом для систем, які використовують Jini. Коли в мережі стає доступним новий сервіс, він реєструється у службі пошуку, тоді клієнт може знайти сервіс, запитавши його у служби пошуку.

Інфраструктура, орієнтована на час виконання, використовує один протокол мережного рівня, або протокол «виявлення» (discovery), і два протоколи об'єктного рівня – відповідно «об'єднання» (join) і «пошук» (lookup). Виявлення дозволяє клієнтам і сервісам знайти службу пошуку; об'єднання - сервісам реєструватися у службі пошуку; пошук призначений для того, щоб клієнт опитував сервіси в пошуках тих, які йому потрібні.

Процес виявлення. Припустімо, наявний Jini-сумісний принтер, який надає послугу друку. Як тільки принтер буде підключено до мережі, він надішле оповіщення присутності з IP-адресою та номером зарезервованого для служби Jini порту, через який принтер може спілкуватися зі службою пошуку.

Сервіс пошуку очікує пакет оповіщення присутності й отримавши, відкриває його та інспектує. Пакет містить інформацію, яка дозволяє службі пошуку визначити, чи потрібно їй зв'язатися з відправником пакета. Якщо це так, то вона з'єднується з відправником безпосередньо, створюючи TCP-з'єднання за IP-адресою і номером порту, отриманими з пакета. Використовуючи RMI, сервіс пошуку надсилає до джерела пакета об'єкт, який називають реєстратор, який забезпечує можливість подальшої взаємодії зі службою пошуку. Викликавши методи цього об'єкта, відправник оповіщувального пакета може об'єднувати і знаходити служби пошуку. У разі наявності Jini-сумісного принтера, який надає послуги друку, служба пошуку має створити TCP-з'єднання

з принтером і надіслати йому об'єкт-реєстратор, за допомогою якого принтер зможе зареєструвати свій сервіс друку через процес об'єднання.

Процес об'єднання. Отримавши об'єкт-реєстратор, постачальник сервісу може виконати об'єднання, тобто стати частиною федерації сервісів, зареєстрованих у службі пошуку. Щоб виконати об'єднання, постачальник сервісу викликає метод *register ()*, який належить об'єкту-реєстратору, передаючи як параметр об'єкт–елемент служби, який є пакетом об'єктів, що описують службу. Метод *register ()* надсилає копію елемента служби сервісу пошуку, де зберігається елемент служби. Після цього постачальник сервісу завершує процес об'єднання: його служба зареєстрована у службі пошуку. Отже клієнти та постачальники послуг спілкуються зі службою пошуку через об’єкт-реєстратор.

Процес пошуку. У разі побудови розподіленої системи служб клієнту потрібно знаходити певні служби, тому після реєстрації у службі пошуку в процесі об'єднання служба стає доступною для використання клієнтами.

Для виконання пошуку клієнт використовує метод *lookup()* об'єкта-реєстратора, для якого як аргумент методу передає шаблон служби – об'єкт, який є критерієм пошуку під час опитування. Шаблон служби може містити посилання на масив об'єктів типу *Class*, які визначають тип (або типи) Java-об'єкта служби, необхідні клієнтові. Клієнт отримує посилання, за яким визначено об'єкт обслуговування як значення, що повертається методу *lookup ()*.

Загалом клієнт шукає сервіс за критерієм тип, який визначається мовою Java, тобто, якщо клієнтові необхідно використати принтер, то він має переслати службі пошуку шаблон сервісу, який містить об'єкт *Class* із інтерфейсом служб друку (всі служби друку мають реалізувати лише цей інтерфейс).

Поділ інтерфейсу і реалізації. Архітектура Jini використовує одну із фундаментальних властивостей Java-об'єктів – поділ інтерфейсу і реалізації. У такому разі обслуговувальний об'єкт може надати клієнтові доступ до служби багатьма способами, зокрема об'єкт може сам являти собою службу, яку завантажує клієнт під час пошуку, а потім усі процеси обробки виконуються локально, а також може лише бути віддаленим компонентом запитуваної служби, при цьому викликаючи методи обслуговувального об'єкта, клієнт надсилає запит мережею до сервера, який і виконує обробку запита.

Важлива властивість архітектури Jini полягає в тому, що мережний протокол, який використовується для спілкування між представником об'єкта й віддаленою службою, не повинен бути відомим клієнту, оскільки він є частиною реалізації служби, тобто розробляється і реалізується розробником служби. Клієнт може спілкуватися зі службою за цим протоколом, оскільки служба вводить свій код (в об'єкті служби) у клієнтський адресний простір. Введений обслуговувальний об'єкт має зв'язуватися зі службою через RMI, CORBA, певний змішаний протокол, побудований на сокетах і потоках, або використовуючи якийсь інший спосіб взаємодії. Клієнту не потрібно використовувати мережний протокол, оскільки він може спілкуватися з інтерфейсом, реалізованим обслуговувальним об'єктом, який застосовує всі необхідні мережні комунікації. З погляду клієнта служба виглядає як інтерфейс, незалежно від того, яким чином ця служба реалізована.

Отже, Jini підвищує абстракцію для програмування розподілених систем від рівня мережного протоколу до рівня інтерфейсів об'єктів. У разі швидкого поширення вбудованих пристроїв, під'єднаних до мережі, багато частин розподіленої системи можуть постачати різні виробники. Jini дає змогу постачальникам застосовувати уніфікацію на рівні інтерфейсів Java, а не на рівні мережного протоколу. Процес

виявлення, приєднання та пошуку забезпечується інфраструктурою Jini, орієнтованою на час виконання, що дозволяє пристроям розшукувати один одного в мережі. Знайшовши один одного, пристрої можуть спілкуватися незалежно за допомогою викликів відповідних методів Java-інтерфейсів.

Модель узгодження. Jini забезпечує як часову, так і незв'язність за посиланнями процесів за допомогою Linda – схожої системи узгодження JavaSpace, тобто це простір розділюваних даних, в якому зберігаються кортежі (типізовані набори посилань на об'єкти Java). В одній системі Jini можуть співіснувати декілька просторів JavaSpace.

Кортежі зберігаються в серіалізованій формі, тобто для збереження кортежу, процес спочатку виконує його маршалінг, причому здійснює маршалінг усіх його полів. Якщо кортеж містить два різні поля, які посилаються на один і той самий об'єкт, то кортеж, що зберігається в реалізації JavaSpace, міститиме дві копії цього об'єкта, над якими проведено процедуру маршалінгу.

Кортеж поміщається у простір JavaSpace за допомогою операції *write*, яка спочатку виконує маршалінг кортежу, а потім зберігає його. Кожного разу під час виконання для кортежу операції *write* в JavaSpace зберігається нова копія цього кортежу, над якою здійснено процедуру маршалінгу (рис.7.3). Можна посилатися на кожну таку копію, як на екземпляр кортежу (*tuple instance*).

Для реалізації генеративного зв'язку в Jini використовують спосіб читання екземплярів кортежу в JavaSpace. Щоб прочитати екземпляр кортежу, процес надає інший кортеж і використовує його як еталон (*template*), який відповідає прочитаним екземплярам кортежу, що зберігаються в JavaSpace. Як і будь-який інший кортеж, еталонний кортеж - типізований набір посилань на об'єкти. У JavaSpace можна прочитати лише екземпляри тих кортежів, які мають однаковий з

еталоном тип. Поля в еталонному кортежі також містять або посилання на реальні об'єкти, або значення *NULL*.

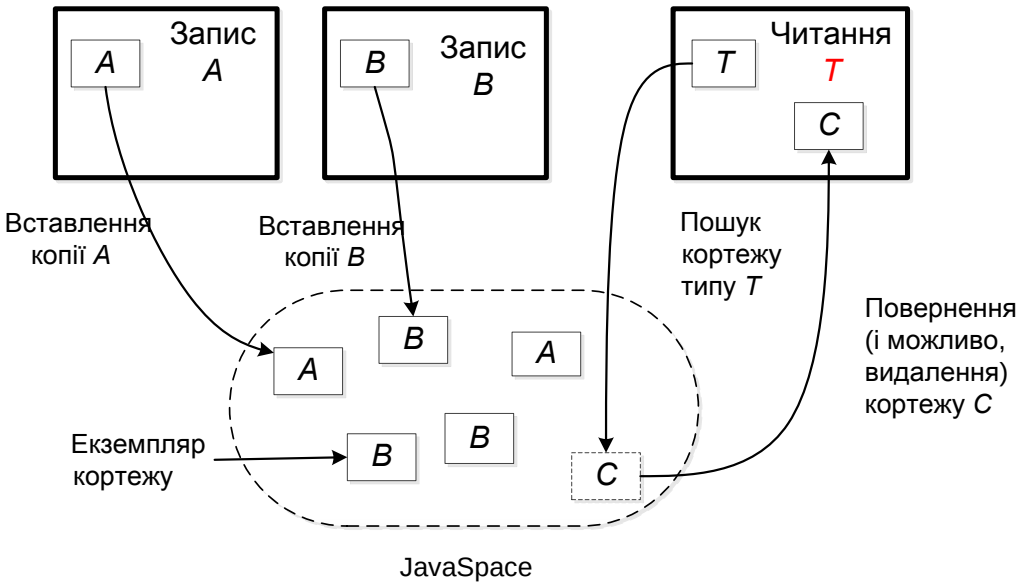


Рис. 7.3. Узагальнена організація простору JavaSpace в Jini

Щоб співставити екземпляр кортежу в JavaSpace з еталонним кортежем, виконується маршалінг еталонного кортежу, включаючи маршалінг полів зі значенням *NULL*. Кожен екземпляр кортежу того ж типу, що й еталон, порівнюється з полями еталонного кортежу, до яких застосовано маршалінг. Два поля збігаються, якщо обидва містять копії одного посилання або якщо поле в еталонному кортежі дорівнює *NULL*, екземпляр кортежу збігається з еталонним кортежем, якщо попарно збігаються відповідні поля.

Коли виявляється екземпляр кортежу, який збігається з еталонним кортежем (частиною операції *read*), виконується демаршалінг цього екземпляра, і він повертається процесу, що ініціював читання. Для читання може бути використана також операція *take*, яка видаляє екземпляр кортежу з простору JavaSpace. Обидві операції блокують процес, який викликав їх, до виявлення потрібного екземпляра кортежу. Максимальний час блокування можна передбачити, а для запобігання значному часу очікування розблокування процесу додатково існують реалізації, які негайно повертають керування, якщо потрібного кортежу не існує.

Порівняно з моделлю публікації/підписки, використовуваною в TIB/Rendezvous, процеси, які застосовують в JavaSpace, не мають виконуватися одночасно, тобто, якщо простір JavaSpace реалізований як сховище, що підлягає зберіганню, всю систему Jini можна вимкнути і запустити знову, не втративши жодного екземпляра кортежу.

Архітектура. JavaSpace складає лише одну із частин системи Jini, тому що, як і TIB/Rendezvous, Jini існує у вигляді компактного набору необхідних засобів і служб, на основі яких можна розробляти розподілене прикладне прогамне забезпечення, яке часто є вільною сукупністю пристроїв, процесів і служб. Уся взаємодія в наявних системах Jini побудована на звертаннях RMI мовою Java.

Архітектуру системи Jini можна подати у вигляді трьох рівнів, як це показано на рис. 7.4. Найнижчий рівень становить інфраструктура, де розміщуються основні механізми Jini, зокрема і ті, які підтримують взаємодію за допомогою звертань RMI мовою Java. Слід окремо відзначити одну важливу властивість моделі Jini: клієнти легко можуть знайти потрібну їм службу. Служби надаються як звичайними процесами, так і пристроями, на яких програмне забезпечення Jini, наприклад віртуальна машина Java, виконуватися не може, тому реєстру вальні та пошукові служби також належать до інфраструктури Jini.

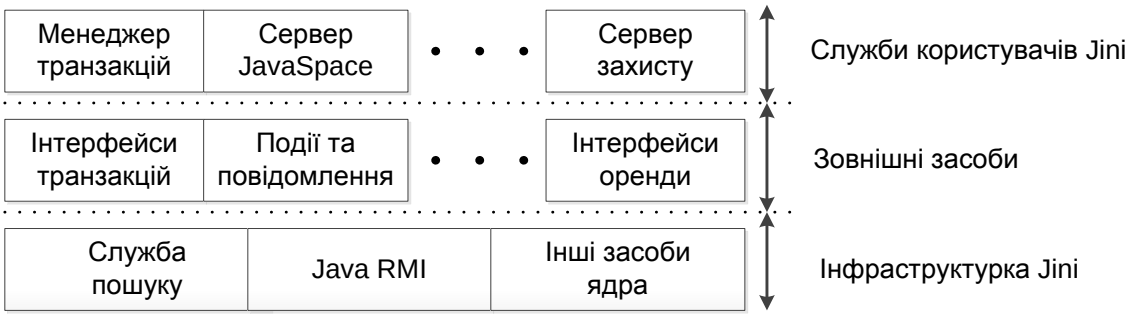


Рис. 7.4. Багаторівнева архітектура системи Jini

Другий (проміжний) рівень утворюють засоби загального призначення, які доповнюють базову інфраструктуру і можуть використовуватися для ефективнішої реалізації служб. До таких засобів

нині належать підсистеми подій і повідомлень, засоби оренди ресурсів і опису стандартних інтерфейсів транзакцій.

Верхній рівень охоплює клієнтів і служби, на відміну від попередніх рівнів Jini, не визначає однозначно складу цього рівня. Нині система підтримує декілька служб верхнього рівня, серед яких сервер JavaSpace і менеджер транзакцій, який реалізує інтерфейси транзакцій Jini. Програмам верхнього рівня дозволяють безпосередньо використовувати механізми інфраструктури Jini.

Приклад. Нехай певна фірма створює корпоративні системи керування та менеджменту. Якщо вона застосує технологію Jini, як найбільш перспективну і безпечну систему розподіленого прикладного програмного забезпечення, то це дозволить їй розробити прикладний шар програмного забезпечення, який забезпечить доступ до розподілених баз даних корпоративної інформації, а також моментальне керування персоналом за допомогою розподіленої системи.

7.4. Порівняння TIB/Rendezvous і Jini

Як TIB/Rendezvous, так і Jini, які є характерними представниками розподілених систем узгодження, націлені на організацію незв'язності процесів за посиланнями, тобто на надання засобів, за допомогою яких процеси могли б попередньо не знати імен тих процесів, з якими матимуть взаємодію.

Перша відмінність між системами полягає у тому, що TIB/Rendezvous незв'язність процесів за посиланням забезпечує за допомогою механізму публікації/підписки, а Jini – генеративним зв'язком через JavaSpace. Додатково Jini забезпечує ще й тимчасову незв'язність процесів.

Інша відмінність між цими системами полягає у тому, що TIB/Rendezvous виконує значну кількість дій щодо взаємодії між процесами, тоді як у Jini система має подати процеси один одному,

після цього взаємодія між ними забезпечується звертаннями RMI мовою Java.

Зв'язок. Взаємодія в TIB/Rendezvous здійснюється передусім за допомогою базового механізму публікації/підписки, тому в цій системі найважливішу роль відіграє групова розсилка. Система застосовує спеціальні заходи, щоб гарантувати успішність групової розсилки й у глобальних мережах. Щоб зв'язок не залежав від прикладного програмного забезпечення, повідомлення мають бути такими, що самовизначаються.

Натомість Jini використовує групову розсилку лише для того, щоб відразу знайти служби пошуку, які потім допомагають клієнтові шукати інші процеси. Будь-яка інша взаємодія, зокрема взаємодія із серверами JavaSpace, реалізується звертаннями RMI мовою Java.

В обох системах події відіграють важливі, але різні ролі. У TIB/Rendezvous механізм подій обслуговує всі взаємодії, зокрема вхідне повідомлення можна отримати лише за умови, що безпосередньо в одержувача встановлено обробника подій для цього повідомлення. З погляду прикладного програмного забезпечення це означає, що для обробки вхідних повідомлень не потрібно блокувальних операцій.

Події в Jini мають іншу природу: кожен процес може запропонувати службу подій, завдяки якій реєструється інший процес для подальшого передавання повідомлень. Коли відбувається певна подія, зареєстрований процес виконує зворотний виклик, який відповідним чином обробляється. Механізм подій Jini – це служба зворотного виклику, яка організовується між парами процесів.

Процеси. Оскільки TIB/Rendezvous і Jini містять лише засоби взаємодії між процесами, то способи організації самих процесів є стандартними, не зважаючи на те, що обидві системи підтримують множину специфічних процесів, які реалізують, наприклад, транзакції,

вони не застосовують ніяких спеціальних інструментальних засобів, щоб зробити ці процеси відмінними від процесів, призначених для сервісів користувача.

Іменування. З погляду іменування між TIB/Rendezvous і Jini наяні такі відмінності:

- імена в TIB/Rendezvous відіграють важливу роль для адресації за темою, всі адреси мають вигляд символьних рядків, схожих на DNS-імена, зокрема дозволено застосовувати прості, але ефективні засоби іменування, включаючи символи-замінники й абревіатури для груп тем.
- порівняно з TIB/Rendezvous, імена в Jini мають вигляд рядків байтів, у яких кожен рядок відповідає об'єкту після маршалінгу, рядки порівнюються лише як числа, хоча можна також використовувати символ-замінник у формі рядка без значення (NULL).
- завдяки ефективності парадигми публікації/підписки TIB/Rendezvous не потребує виділеної служби іменування, яка б перетворювала імена в адреси процесів, натомість система Jini має підтримувати виділену службу пошуку, що дозволяє клієнтові локалізувати процес за іменем на основі атрибутів (це ім'я у вигляді послідовності рядків байтів).

Синхронізація. Системи TIB/Rendezvous і Jini підтримують механізми транзакцій, але використовують різні моделі транзакцій. У TIB/Rendezvous транзакції об'єднують послідовності операцій публікації та отримання повідомлень (а також операцій з базами даних) у єдину транзакцію. Транзакції (точніше, транзакційний обмін повідомленнями) здійснюються на спеціальному рівні, на якому перебуває розширення звичайної бібліотеки публікації/підписки, із застосуванням менеджера транзакцій. TIB/Rendezvous контролює, щоб транзакції у процесі обміну повідомленнями задовольняли властивостям ACID, тобто атомарності, несуперечливості, ізольованості й довговічності транзакції.

Jini містить лише протокол транзакцій, який дозволяє об'єднати у транзакцію декілька операцій, викликаних клієнтом, але забезпечити дотримання властивостей ACID має процес, який ініціював її. Спеціальний менеджер транзакцій Jini гарантує, що спільно із серверами JavaSpace забезпечить дотримання властивостей ACID для транзакцій.

Робота транзакції в TIB/Rendezvous обмежена одним процесом. Якщо у транзакції беруть участь декілька процесів, то необхідний традиційний менеджер транзакцій, який міг би обслуговувати не лише обмін повідомленнями, але й паралельну обробку процесів. Натомість у Jini в одній транзакції можуть брати участь декілька клієнтських процесів, хоча зазвичай ініціативу з організації і завершення транзакції проявляє тільки один клієнт.

Окрім транзакцій Jini, на відміну від TIB/Rendezvous (де немає ніяких інших механізмів синхронізації, окрім транзакцій), підтримує також механізм синхронізації на основі блокувальних операцій JavaSpace, де особливо ефективно реалізовано розподілені блокування.

Кешування і реплікація. Ні TIB/Rendezvous, ні Jini не підтримують кешування і реплікацію, це має реалізовувати прикладне програмне забезпечення.

Відмовостійкість. Обидві системи використовують надійний зв'язок, причому TIB/Rendezvous підтримує зв'язок, який підлягає зберіганню як у формі сертифікованої доставки повідомлень, так і у формі транзакційного обміну повідомленнями, тоді як Jini застосовує засоби надійності, які використовують звертання RMI мовою Java.

У TIB/Rendezvous відмовостійкість забезпечується підтримкою груп процесів на проміжному рівні, а Jini не має спеціальних засобів підтримки груп процесів. Передбачено, що у разі потреби цю підтримку реалізовуватимуть прикладні програми. Жодна з систем не

має засобів відновлення після відмов, за винятком реалізованих як частина транзакцій.

Захист. TIB/Rendezvous підтримує захист у формі окремого протоколу організації захищеного каналу між видавцем і підписником. Контроль доступу здійснюють прикладні програми, використовуючи TIB/Rendezvous як проміжний рівень взаємодії.

У Jini захист цілком забезпечується тими стандартними засобами захисту, які зазвичай застосовують в системах на основі Java, контролюють доступ на рівні класів, а також автентифікують і авторизують користувачів, застосовуючи виділений сервер захисту JAAS (Java Authentication and Authorization Service).

7.5. Висновки

1. Розподілені системи узгодження суттєво впливають на розробку розподіленого прикладного програмного забезпечення. Більшості таких систем характерна відсутність зв'язності процесів, яка визначається посиланнями, тобто процеси для зв'язку один з другим не потребують явних посилань, а також відсутня тимчасова зв'язність, коли для взаємодії процесів не обов'язково, щоб вони виконувалися одночасно.
2. Одна з важливих груп систем узгодження – це системи, які ґрунтуються на парадигмі видавця/підписника, зокрема TIB/Rendezvous. У цій моделі повідомлення доставляються одержувачам не відповідно до їх адрес, а відповідно до теми повідомлень. Процеси, охочі отримувати повідомлення, мають підписатися на певну тему. За вибір шляху повідомлень від видавця до підписника відповідає проміжний рівень.
3. Група систем узгодження, яка застосовує поєднання не зв'язаних за часом і за посиланнями процесів, тобто використовує модель генеративного зв'язку, вперше запропоновану в системі Linda.

4. Генеративний зв'язок реалізують простори кортежів колективного доступу. Кортеж – це типова структура даних, схожа на записи. Щоб витягувати з простору кортежів потрібний кортеж, процес відшукує його за допомогою еталонного кортежу. Кортеж, який відповідає еталонному, визначається і повертається процесу, який запитав його. Якщо збігів немає, то процес блокується.

5. Системи узгодження відрізняються від більшості інших розподілених систем тим, що вони націлені на надання зручного способу зв'язку між процесами, які нічого не знають один про одного заздалегідь. Зв'язок може і далі здійснюватися зі збереженням анонімності. Основна перевага такого підходу – його гнучкість, тобто можливість доповнювати і змінювати систему, яка продовжує працювати.

6. Принципи розподілених систем так само застосовні й до систем узгодження, хоча кешування і реплікація відіграють у сучасних їх реалізаціях не таку важливу роль, а іменування в них істотно пов'язано з пошуком за атрибутами. Аналогічний підхід реалізовано й у службах каталогів.

7.6. Запитання для самоконтролю

1. Який основний підхід використовується в системах узгодження ?
2. Як реалізується пряме узгодженням (direct coordination)?
3. Як реалізується узгодженням за допомогою поштової скриньки (mailbox coordination)?
4. Як реалізується узгодження на зустрічі (meeting-oriented coordination)?
5. Як узгоджуються процеси в системах публікації/підписки (publish/subscribe systems)?

6. Як реалізується узгодження з генеративним зв'язком (generative communication)?

7. Який принцип побудови системи TIB/Rendezvous?

8. Яку модель узгодження застосовано у системі TIB/Rendezvous?

9. Яку архітектуру має система TIB/Rendezvous?

10. Який принцип побудови системи Jini?

11. Яку модель узгодження застосовано у системі Jini?

12. Яку архітектуру має система Jini?

13. Які відмінності за зв’язком мають системи TIB/Rendezvous і Jini?

14. Які відмінності за процесами мають системи TIB/Rendezvous і Jini?

15. Які відмінності за іменуванням мають системи TIB/Rendezvous і Jini?

16. Які відмінності за синхронізацією мають системи TIB/Rendezvous і Jini?

17. Чи мають відмінності за за кешуванням і реплікацією системи TIB/Rendezvous і Jini?

18. Які відмінності за відмовостійкістю мають системи TIB/Rendezvous і Jini?

19. Які відмінності за захистом мають системи TIB/Rendezvous і Jini?

8. ПОШУКОВІ СИСТЕМИ

8.1. Значення пошукових систем

Згідно з даними компанії Netcraft, станом на жовтень 2010 року в глобальній мережі інтернет зареєстровано 232 839 963 сайтів. Простий пошук за досить поширеним ключовим словом дає зазвичай від десятків тисяч до декількох мільйонів посилань. Очевидно, що робота з такою великою кількістю документів неможлива, тим більше, що переважна їх частина нерелевантні, тобто містять інформацію, яка не стосується запиту.

Розрізняють низку факторів, які пошукові системи використовують під час ранжирування сайтів у результатах видачі запиту. Алгоритми ранжирування весь час доповнюються й оптимізуються, тому, розробляючи web-ресурс, варто враховувати актуальні фактори, які впливають на рейтинг web-сайту.

Сервіс статистики Bigmir.net подає максимально точну інформацію щодо популярності пошукових систем в Україні, зокрема ринок пошукових запитів у березні 2010 року поділено таким чином: Google — 73.9 %, Яндекс — 20.4 %, Ukr.net — 2.3 %, Search.Mail.ru — 1.0 %, Meta.ua — 0.6 %, Bigmir.net — 0.5 %.

Відповідно до даних статистики, актуальними для розгляду є методи оптимізації web-ресурсів з урахуванням способів ранжування пошукової системи Google. Проте варто відзначити, що в разі правильного просування сайту в пошуковій системі Google, позиції у видачі інших пошукових систем також підвищуються.

Для організації ефективного пошуку в мережі Internet необхідно виконати низку операцій щодо наповнення кожної сторінки, розміщеної в мережі. Цей процес називають Search Engine Optimization

(SEO), що в перекладі означає оптимізація під пошукові системи або просто пошукова оптимізація.

Як і більшість сучасних понять, SEO має кілька визначень, проте загалом під пошуковою оптимізацією розуміють процес роботи із сайтом, його внутрішніми факторами, що впливають на ранжування в пошукових системах, – структурою, контентом, кодом HTML; зовнішніми факторами ранжування – посиланнями на сайт для підвищення релевантності ресурсу певними, заздалегідь відомими ключовими словами, популярності сайту для пошукових машин і, відповідно, позицій у пошукових результатах для залучення більшої кількості відвідувачів на сайт.

8.2. Історія розвитку пошукових систем

Першою пошуковою системою для Всесвітньої павутини був «Wandex», вже не існуючий індекс, який створював «World Wide Web Wanderer» – бот, розроблений Метью Грей (Matthew Gray) з Массачусетського технологічного інституту в 1993 році. Того ж року з'явилася пошукова система «Aliweb», яка працює досі. Першою повнотекстовою («crawler-based» - індексує ресурси за допомогою робота) пошуковою системою стала «WebCrawler», запущена 1994 року. На відміну від попередніх систем, вона дозволяла користувачам шукати за будь-якими ключовими словами на будь-яких web-сторінках. Такий підхід став стандартним для всіх основних пошукових систем.

З 1996 року Ларрі Пейдж і Сергій Брін розробляли пошукову систему BackRub, а 1998 року на її основі створили пошукову систему Google.

8.3. Архітектура пошукової системи

У пошуковій системі Google, які будь-якій пошуковій системі, можна виокремити такі основні частини:

– робот (краулер, спайдер, індексатор) - відповідає за збирання інформації, тобто емулює роботу користувача, завантажуючи сторінки і зберігаючи їх у базі даних;

– база даних, у якій зберігається і сортується зібрана роботом інформація;

– клієнт - у цій частині обробляються запити користувачів. Клієнт може бути рознесений по декількох комп'ютерах, які фізично не зв'язані, але мають доступ до бази даних.

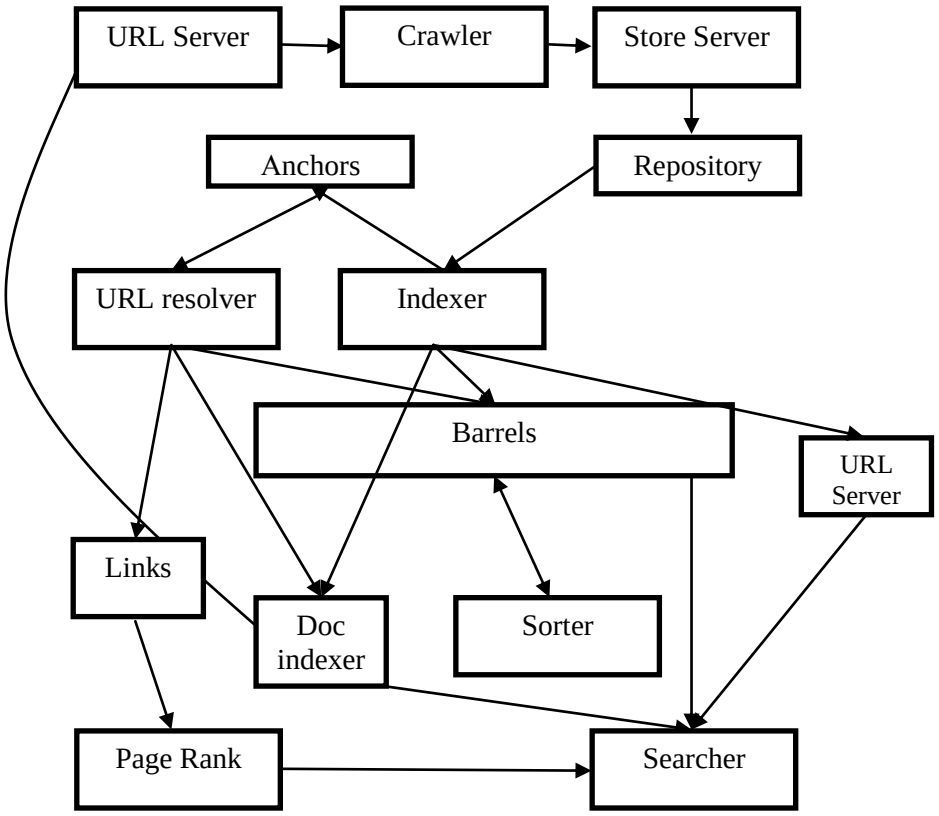


Рис. 8.1. Структура пошукової системи Google

Пошукова система Google для здійснення пошукових запитів використовує такі програмні компоненти (рис. 8.1):

– URL Server, який відповідає за ведення списку всіх адрес;

– Crawler - робот, який завантажує сторінки зі списку адрес і передає в Store Server;

– Store Server, який зберігає сторінки в Repository, найчастіше у вигляді HTML-документа, при цьому вся додаткова інформація (зображення, flash-анімація тощо) не зберігається;

– Indexer, який перетворює збережені в Repository HTML-документи на послідовність слів і зберігає їх у Barrles (база даних);

– Lexicon, який містить список усіх слів, які найчастіше зберігаються в таблиці з двома полями, що мають назву «номер» і «слово», за рахунок чого досягається економія місця в базі даних, оскільки довгі слова замінюються досить коротким номером;

– Anchors, який містить виокремлені компонентом Indexer посилання (URL);

– URL Resolver, який здійснює обробку URL, якщо знаходяться нові посилання, то вони передаються компоненту URL Server;

– Links, що визначає, які сайти на які посилаються, і передає інформацію до PageRank;

– PageRank, який визначає рейтинг сайту за основним критерієм - кількістю посилань на цей сайт;

– Searcher, який є клієнтом, що найчастіше користується статичною базою даних, яка оновлюється приблизно раз на добу.

8.4. Прямий і зворотний індекс

Для кожного запису в базі даних зберігається певна додаткова інформація, зокрема вона містить прямий і зворотний індекси.

У разі ідентифікації записів бази даних прямим індексом записи відсортовано за номером документа, тобто для кожного запису зберігають відповідний номеру список слів і для кожного слова - перші кілька (наприклад, вісім) позицій входження слова в документ,

кількість входжень і формат входження. Під форматом входження розуміють входження слова в тексті, в описі до зображення, в заголовку тощо, ці слова матимуть пріоритет під час пошуку. Прямий індекс оновлюється постійно у процесі функціонування робота, тому для кожної сторінки в базі даних зберігається частота передбачуваного оновлення, яку визначають під час чергового входження робота на сторінку. Якщо оновлення відсутні, то частота збільшується удвічі, якщо ж сторінка за цей період часу змінювалася, то частота зменшується. Також варто відзначити, що найчастіше робот індексує не всі слова з документа (наприклад, лише першу тисячу слів) та не всі документи з одного сайта.

Зворотний індекс використовує клієнт під час пошуку, коли записи сортуються за словами. Для кожного запису зберігається номер слова, список документів, які містять це слово, і повна інформація щодо його входження в певні документи. Зворотний індекс оновлюється не так часто, як прямий, а приблизно раз на добу.

Алгоритм функціонування клієнта під час пошуку інформації за ключовими словами. Спочатку запит розбивається на слова, далі видаляються «стоп» - слова, тобто слова, які трапляються майже в усіх документах. На наступному кроці для кожного слова із запиту визначають відповідність його номеру номеру слова із словника, який формує «лексику» даного сайту, після чого знаходять у зворотному індексі список документів, які містять це слово. Із цих списків створюється новий, який містить лише ті документи, які входили до списків для всіх слів. Потім, на основі визначених характеристик для кожного з документів обчислюється ступінь релевантності, і список сортується за цією ознакою. На цьому кроці для всіх документів створюються анотації, тобто зміст тега «*description*», контекст входження слів із запиту (наприклад, слова розташовані найближче або це є перше входження), перше речення або заголовок документа.

8.5. Визначення PageRank

8.5.1. Загальні відомості

У разі вибору зі списку документів, що мають бути отримані за деяким запитом, потрібні документи визначають за критерієм їх відповідності запиту, що включає й завдання обчислення «важливості» документа, яка враховує такі парметри як частота оновлення сторінки, кількість відвідувань сторінки та наявність реєстрації в каталозі данної пошукової системи. Ці параметри легко підробити (крім реєстрації в каталозі), тому пошукові системи вводять низку характеристик, які не можна підробити.

Для визначення «важливості» документа або сторінки в сучасних пошукових системах використовують спеціалізовані засоби, зокрема Google застосовує метод **PageRank,** який визначає певне місце документа серед інших (його ранг).

Коли всі інші фактори, зокрема тег Title і ключові слова, враховано, Google використовує значення PageRank, щоб відкорегувати результати так, щоб більш «важливі» сайти піднялися відповідно вгору на сторінці результатів пошуку користувача.

Узагальнений алгорим ранжирування в Google такий:

1. Знайти всі сторінки, які відповідають ключовим словам пошуку.
2. Упорядкувати відповідно до «сторінкових факторів», наприклад ключових слів.
3. Урахувати текст посилань на сторінки.
4. Відкорегувати результати пошуку за даними PageRank.

8.5.2. Обчилення PageRank

Для обчислення значення PageRank для сторінок необхідно змоделювати рух користувача мережею, використовуючи модель випадкового блукання, за якою мережу зображають у вигляді орієнтованого графа $G = (X, T)$, де $X = \{x_i\}$ – множина вершин, які є документами, а $T = \{t_{ij}\}$ – множина ребер, які є посиланнями із сторінки x_i на сторінку x_j. Користувач починає свій рух від випадкової вершини x_i, далі на кожному кроці він з імовірністю ε (зазвичай ε дорівнює близько 0.15) переходить у випадкову вершину x_k і з імовірністю $1-\varepsilon$ переміщується по одному з ребер, які ведуть від цієї вершини.

Значення PageRank (PR_k) є ймовірністю перебування користувача у x_i-й вершині через k кроків.

Якщо кількість вершин k нескінченна ($lim\ (k) \rightarrow \infty$), то ймовірність перебування користувача у i-й вершині через k кроків дорівнюватиме $PR_k(x_i) = PR(x_i)$, тобто для кожної вершини є гранична ймовірність знаходження в ній.

Розглянемо основне рівняння для обчислення значення PageRank, дя чого введемо такі позначення:

$x_1, .., x_n$ - вершини, з яких прямують ребра t_{ij} із вершини x_i у вершину x_j,

$C(x_i)$ - кількість ребер, що виходять із вершини x_i,

N – загальна кількість вершин.

Для розрахунку ймовірності перебування користувача у i-й вершині через k кроків використовують такий вираз:

$$PR(x_i) = \varepsilon/N + (1 - \varepsilon).$$

На практиці точно розрахований PageRank не використовують, а замість $PR(x_i)$ беруть $PR_{50}(x_i)$, тобто після розрахунку PageRank його значення округляється так, щоб це було ціле число від 1 до 10, яке називають ваговим коефіцієнтом (вагою) PageRank для сторінки.

Коли Google був тільки дослідницьким проектом, його розробники написали статтю, у якій описано формулу, яка визначає вагу для сторінки.

$PR(A) = (1-d) + d\ (PR(T_1)/C(T_1) + ... + PR(T_n)/C(T_n))$,

де $PR(A)$ – вага PageRank сторінки *A* (обчислювана вага);

d – коефіцієнт загасання, який зазвичай дорівнює 0,85;

$PR(T_1)$ – вага PageRank сторінки, яка посилається на сторінку *A*;

$C(T_1)$ – кількість посилань із цієї сторінки;

$PR(T_n)/C(T_n)$ – відношення, яке враховує те, що розрахунок виконується для кожної сторінки, яка посилається на сторінку *A*.

Щоб обчислити вагу PageRank сторінки *A*, необхідно знати вагу PageRank усіх сторінок, які посилаються на сторінку *A*. Їх вага PageRank частково залежатиме від сторінки *A*, яка посилається на них, або якихось інших сторінок, які також посилаються на них.

Якщо обчислюється вага PageRank сторінки *A*, на яку посилається сторінка *B*, то вага сторінки PageRank *B* «передається» (додається під час обчислення) сторінці *A*, частково зменшуючись при цьому. Так відбувається з кожним наступним посиланням, проте загальна вага буде завжди однаковою.

Приклад реалізації зображено на рис. 8.2, який показує послідовність обміну вагою PageRank між сторінками у разі обчислень вагових коефіцієнтів кожної із сторінок. З рис. 8.2 видно, що вага сторінки *A* частково враховуватиметься у вазі сторінки *B*, оскільки вона на неї посилається, а вага сторінки *C* частково врахує вагу сторінок *A* і *B*, оскільки вони посилаються на неї. Таким чином, сторінка *A* дозволить двічі врахувати свою вагу під час розрахунку ваги сторінки *C* за рахунок наявності посилань. Такий коефіцієнт ваги (вагу) кожної сторінки *PR(Name)* називають MiniRank, розрахунок якого дуже схожий на розрахунок PageRank.

Нехай ваги MiniRank у цих сторінок дорівнюють одиницям, тоді діаграма має вигляд, як зображено на рис. 8.2.

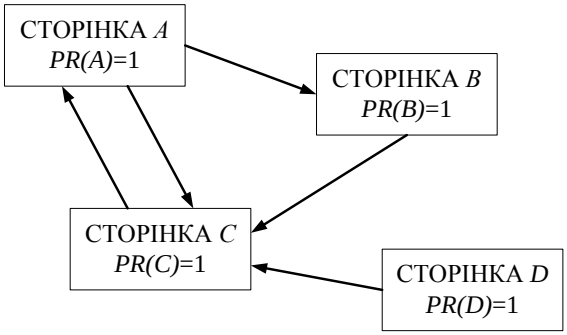


Рис. 8.2. Діаграма із заданими вагами MiniRank

Відмінність розрахунку MiniRank полягає у тому, що спочатку застосовують коефіцієнт згасання, який свідчить про те, що сторінка не може враховувати вагу іншої, яка на неї посилається так, щоб інша сторінка була настільки ж важлива, як вона сама. Потім збережену вагу ділять на кількість посилань, далі обчислюється підсумкова вага, яка має додаватися до всіх сторінок перед остаточним розрахунком.

Для сторінки *A* значення ваги MiniRank доступне для врахування після згасання дорівнює *1×0,85=0,85*. Зі сторінки *A* виконується два посилання, тому по закінченні ітерації додано 0,425 до ваги MiniRank сторінки *B* і 0,425 до ваги MiniRank сторінки *C*. Сторінки *B*, *C* містять лише одне посилання, тому сторінка *B* передасть значення ваги, яке дорівнюватиме 1×0,85=0,85, сторінці *C* після обчислення всіх посилань, так само сторінка *C* - вагу 1×0,85=0,85 сторінці *A*, а сторінка *D* - сторінці *C*. Результати операцій зображені на рис. 8.3.

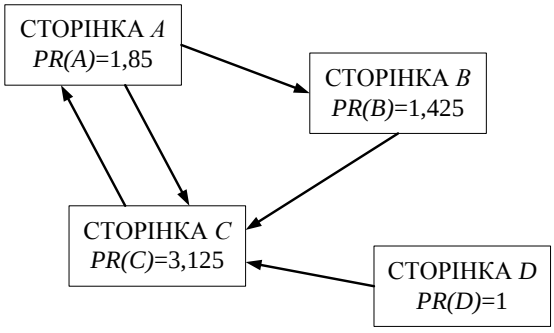


Рис. 8.3. Діаграма після обчислень ваг MiniRank

Нові значення ваги MiniRank показують, наскільки важлива сторінка *C*, але оскільки всі сторінки спочатку мали однакові значення, то розрахунок визначає тільки популярність у посиланнях (link popularity). Суть PageRank й MiniRank полягає тільки в тому, що сторінкам, на які частіше посилаються, варто одержати більшу вагу, тому необхідно виконати те саме ще раз, тоді сторінка *C* матиме суттєвіший вплив, тому що значення її поточної ваги MiniRank стане вищим.

Значення поточної ваги MiniRank сторінки *A* дорівнює 1,85, а значення MiniRank, доступне для врахування в інших сторінках, після застосування згасання становить 1,85×0,85=1,5725. Є два посилання зі сторінки, тому по завершенні ітерації додамо 0,78625 до значень ваги MiniRank сторінки *B* і ваги MiniRank сторінки *C*.

У сторінки *B* є тільки одне посилання, тому вона передасть 1,425×0,85=1,21125 сторінці *C* після завершення всіх обчислень з посиланнями. Сторінка *C* також має одне посилання, але при цьому має значну вагу 3,125 MiniRank, тому вона передасть 3,125×0,85=2,65625 сторінці *A*. Сторінка *D* має одне посилання, тому вона передає 0,85 сторінці *C*. Результати обчислень показано на рис. 8.4.

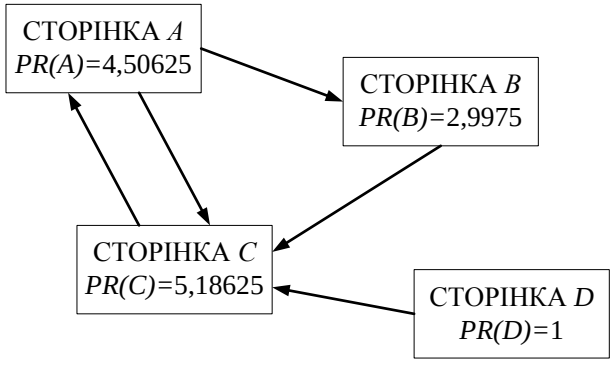


Рис. 8.4. Діаграма результатів після другої ітерації обчислення значень ваги MiniRank

Як бачимо, сторінка *C* має найбільшу вагу MiniRank, сторінка *A* – наступну за значенням. На практиці потрібно було б повторити ці дії 50…100 разів, щоб гарантувати високу точність попередніх ітерацій.

Зворотний зв'язок PageRank. Розглянемо модифікацію алгоритму реалізації MiniRank (див. рис. 8.2). Під час першої ітерації обчислень сторінка *C* додає значеню ваги сторінки *A* збільшення значення ваги MiniRank (PageRank), а під час наступної у неї самої збільшується значення ваги, пропорційно новому значеню ваги сторінки *A*, яка покращилася (вона одержує назад частину свого значення ваги MiniRank). Це зворотний зв'язок PageRank – невід'ємна частина системи.

8.6. Пошукова оптимізація

8.6.1. Фактори, які впливають на ранжування web-сайтів

У процесі ранжирування web-сайтів пошукова система Google використовує близько 200 критеріїв, частину яких наведено нижче:

I. Доменне ім'я:

1) вік;
2) дата реєстрації;
3) публічність/прихованість інформації про домен;
4) географічна прив’язка доменного імені;
5) домен першого рівня;
6) рівень доменного імені;
7) як часто змінювалася IP-адреса, прив'язана до домену;
8) як часто змінювався власник;
9) ключові слова в імені домену;
10) IP-адреса;
11) сайти, які містяться за цією ж IP-адресою;

12) згадування доменного імені (не зовнішні посилання);

13) географічна прив’язка, зазначена в Google Webmaster Tools.

II. Сервер:

1) географічне розташування сервера;
2) доступність сервера (uptime).

III. Архітектура сайта:

1) URL-архітектура;
2) HTML і CSS структура;
3) семантичність верстання;
4) використання зовнішніх CSS/JS файлів;
5) доступність внутрішньої навігації (наприклад, якщо відключено JavaScript та інше);
6) ієрархічні URL;
7) валідність HTML і CSS верстки;
8) використання cookies.

IV. Контент:

1) мова;
2) унікальність;
3) обсяг тексту;
4) обсяг тексту без посилань;
5) обсяг чистого тексту (без посилань, зображень та ін.);
6) актуальність контенту, орієнтованість на поточний попит;
7) семантичність контенту;
8) тип контенту (діловий, інформаційний, навігаційний);
9) відповідність контенту ринковій ніші;
10) використання певних ключових фраз (азартні ігри, знайомства та ін.);
11) підписи до зображень;
12) шкідливий контент (наприклад, на зламаних сайтах);

13) граматика і пунктуація;

14) використання унікальних, нових фраз.

V. Якість внутрішніх посилань:

1) кількість внутрішніх посилань;

2) кількість внутрішніх посилань на сторінки, що містять контент, релевантний тексту в посиланні;

3) кількість посилань на сторінку з контенту сайта (не з навігаційних панелей);

4) кількість посилань, що використовують атрибут nofollow;

5) щільність внутрішніх посилань.

VI. Сайт:

1) контент файлу robots.txt;

2) загальна частота оновлення сайта;

3) кількість сторінок сайта;

4) вік сторінок сайта, з моменту першої індексації Google;

5) XML Sitemap;

6) довірча інформація (контактна інформація, угоди, публічні оферти тощо);

7) тип сайта (блог, інформаційний, комерційний та ін.).

VII. Web-сторінка:

1) тег meta з атрибутом robots;

2) вік сторінки;

3) частота оновлення;

4) унікальність контенту всередині сайта;

5) легкість читання;

6) швидкість завантаження сторінки;

7) рівень вкладеності сторінки;

8) кількість внутрішніх посилань сайта на цю сторінку;

9) кількість зовнішніх посилань сайта на цю сторінку.

VIII. Ключові слова:

1) ключові слова в *TITLE*;
2) близькість до початку в *TITLE*;
3) в *alt*-атрибутах;
4) у тексті внутрішніх посилань;
5) у тексті зовнішніх посилань;
6) ключові слова, виділені жирним шрифтом і курсивом;
7) ключові слова, що містяться на початку контенту сайту;
8) щільність ключових слів;
9) синоніми до ключових слів;
10) ключові слова в назві файлів;
11) ключові слова в URL;
12) відсутність непотрібних (нетематичних) ключових слів;
13) використання, зловживання ключовими словами у HTML-коментарях.

IX. Зовнішні посилання з сайта:

1) кількість зовнішніх посилань з домену;
2) кількість зовнішніх посилань зі сторінки;
3) якість сторінок, на які посилається сайт;
4) зовнішні посилання на «поганих сусідів»;
5) релевантність зовнішніх посилань;
6) посилання на неіснуючі сторінки (404-статус);
7) посилання на SEO-агентства (для сайтів клієнтів);
8) зовнішні посилання на зображення.

Зовнішні посилання на сайт:

1) релевантність сайта;
2) релевантність сторінки;
3) якість сайта;
4) якість сторінки;
5) посилання із сітки сайтів;

6) цитування (схожість зовнішніх посилань);

7) посилання:

- нормативний текст;
- різні IP-адреси сайтів, які посилаються;
- географічна посилальна різноманітність;
- посилання з різних доменів верхнього рівня;
- тематичне розмаїття;
- тип сайта, який посилається (каталог, ЗМІ та ін.);
- місце, в якому розташовано посилання (підвал, контент тощо);

8) трастовість сайта;

9) зворотні посилання від «поганих сусідів» або від сайтів, позначених як неякісні;

10) взаємні посилання;

11) посилання із соціальних мереж;

12) тренд збільшення посилань;

13) цитування сайта у Вікіпедії, Dmoz;

14) нормативний профіль сайта (продавав/купував посилання та ін.);

15) посилання із закладок.

X. Кожне зовнішнє посилання:

1) трастовість домену верхнього рівня;
2) трастовість основного домену;
3) трастовість сторінки;
4) місцерозташування посилання (підвал, контент тощо);
5) нормативний текст (alt-атрибут для посилань-зображень);
6) title-атрибут посилання.

XI. Профілі відвідувачів сайту:

1) кількість відвідувачів;
2) географія відвідувачів;
3) показник відмов;
4) звички користувачів;

5) тренд відвідуваності;
6) як часто переходять на сайт під час виконання пошукого запиту.

XII. Песимізація і фільтрація:

1) перенасичення ключовими фразами;
2) купівля посилань для сайта;
3) продаж посилань на сайті;
4) спам (у коментарях, форумах та ін.);
5) наявність значної кількості непотрібної інформації;
6) прихований текст;
7) не унікальний контент;
8) історія песимізації домену;
9) історія песимізації власника;
10) історія песимізації параметрів власника;
11) останні записи хакерів;
12) подвійні й приховані перенапрямлення на інші сайти.

XIII. Інше:

1) дата реєстрації домену в Google Webmaster Tools;
2) згадування сайта в Google News;
3) згадування сайта в Google Blog Search;
4) використання сайта в Google AdWords;
5) використання сайтом сервісу Google Analytics;
6) кількість згадувань назви сайта.

Ознайомившись зі списком чинників, можна дійти висновку, що розвиток пошукових мереж спрямовано на врахування соціального «людського» контенту.

8.6.2. Етапи пошукової оптимізації

Процес оптимізації (удосконалення) web-сайтів під алгоритми роботи пошукових систем може включати ряд етапів, які виконують у довільному порядку з метою найбільш ефективної їх відповідності вимогам пошукових запитів користувачів.

1. **Визначення цілей.** На цьому етапі необхідно визначити загальну мету просування web-сайта, цілі якого такі:

– наявність сайта у результатах роботи пошукової системи, які подаються як список посилань;

– високі позиції в пошукових системах за певними запитами;

– перевищення рейтингу конкурентів у позиціях за деякими запитами;

– підвищення відвідуваності ресурсу (трафіку);

– пошук цільової аудиторії та постійних зацікавлених клієнтів;

– підвищення коефіцієнта конверсії та, відповідно, прибутковості від реклами або партнерської програми;

– інші цілі.

Залежно від обраних цілей слід формувати власний план просування і методи отримання звітів і аналізу результатів пошукової оптимізації.

Попередній етап. На цьому етапі аналізують тематичний сегмент сайта (майбутній зміст), шукають нішу позиціонування (як подати сайт), створюють семантичне ядро запитів, визначаючи основні запити, за якими необхідно потрапляти в першу десятку, підбираючи синоніми для бажаних запитів.

Аналіз конкуренції за кожним із запитів. Після побудови ядра запитів, необхідно проаналізувати ринок і визначити головних конкурентів, їх методи пошукової оптимізації, позиції й тенденції. Найчастіше після аналізу дій конкурентів необхідно вносити зміни до плану просування сайта.

Оптимізація під аудиторію. Основну частину цього етапу становить робота зі змістом: слід звернути увагу, по-перше, на тексти, оскільки індексація виконується саме за словами, по-друге, на наявність на сайті додаткових сервісів, які привернуть увагу додаткової аудиторії.

Оптимізація під пошукові запити полягає у підборі сторінок під кожну групу запитів, тому що просування за всіма запитами однієї головної сторінки сайта є складним завданням, через що для цього підбирають певні сторінки і кожну з них налаштовують під визначену ключову фразу з ядра запитів. Головну сторінку варто налаштовувати під високочастотні, у той час як сторінки другого рівня - під середньочастотні запити.

Підготовка наповнення сторінок. На цьому етапі виконують підготовку сайта до індексації, яка охоплює такі складові:

1. Підготовка текстів. Оптимальними слід уважати сторінки, які містять 500…3000 слів або 2…20 кб тексту (2…20 тис. символів), оскільки сторінка лише з декількома пропозиціями, має менше шансів потрапити в топ пошукових систем. Крім того, значний обсяг тексту на сторінці підвищує видимість сторінки в пошукових системах за рахунок рідкісних або випадкових пошукових фраз, яка зумовлює приплив відвідувачів.

2. Кількість ключових слів на сторінці. Ключові слова (фрази) мають траплятися в тексті як мінімум 3…4 рази, а верхня межа їх появи залежить від загального обсягу сторінки: чим він більший, тим більше повторень можна зробити.

Окремо слід розглянути використання пошукових фраз, тобто словосполучень, які містять декілька ключових слів. Найкращих результатів пошуку можна досягти, якщо фразу повторювати в тексті кілька разів в повному обсязі та всі її слова у встановленому порядку, а також повторювати кілька разів в тексті окремі слова із фрази. Крім

цього, має бути деяка відмінність (розбалансування) між кількістю входжень кожного зі слів - складових фрази.

Приклад. Потрібно оптимізувати сторінку під фразу «dvd програвач», для цього таку фразу повторюють в тексті десять разів, слово «dvd» - окремо ще сім разів, слово «програвач» - ще п’ять разів.

3. Щільність ключових слів на сторінці, вимірювана у відсотках, показує відносну частоту входження слова в текст. Наприклад, якщо задане слово повторюється п’ять разів на сторінці зі 100 слів, то щільність цього слова дорівнює 5%. Через дуже низьку щільність пошукова система не надасть належного значення цьому слову, а занадто висока - здатна використати спам-фільтр пошукової системи (тобто сторінку буде штучно знижено в результатах пошуку через надмірно часте вживання ключової фрази). Оптимальною вважають щільність ключового тексту 5...7 %. Якщо фраза містить декілька слів, то слід обчислити сумарну щільність усіх ключових слів (складових фрази) і переконатися, що вона укладається у вказані межі.

Практика показує, що щільність ключового тексту вища за 7…8%, хоч і не призводить до яких-небудь негативних наслідків, але і не приносить позитивного результату .

4. Заборона індексації деяких сторінок. Цю операцію здійснюють для того, щоб сайт не з’являвся у результатах за непотрібними словами.

Приклад. Великий сайт-каталог, у якому, зокрема, був опис телефону Nokia, знаходили саме за запитом про цей телефон, що не передбачали розробники сайту.

5. Тег «TITLE» - один з найбільш важливих тегів, якому пошукові системи надають істотного значення, тому бажано використовувати ключові слова у змісті тегу «*TITLE»*. Через це посилання на сайт під час надання користувачеві пошуковою системою міститиме текст із тегу *TITLE*, утворюючи деякою мірою «візитну картку» сторінки. Саме за цим посиланням відвідувач пошукової системи переходить на потрібний йому сайт, тому тег «*TITLE»* має не лише містити ключові слова, але й бути інформативним і привабливим. Зазвичай, у список

надання посилань користувачеві пошуковою системою потрапляє 50…80 символів з тегу «*TITLE»*, тому розмір заголовка бажано обмежити цією кількістю.

Приклад. *TITLE*-тег в HTML-коді сайта: <*TITLE*> просування сайта: розкручування сайтів, пошукова оптимізація </*TITLE*>.

«TITLE»-теги мають містити основні ключові слова, які використовуються на сторінці, що просувається, і кожна сторінка сайта повинна мати унікальний «*TITLE»*-тег.

Для того, щоб сторінка мала суттєвий рейтинг під час виконання пошуку варто уникати у заголовному тезі «стоп-слів», тобто слів, які стали настільки загальними, що пошукові машини ігнорують їх або виводять досить некоректні результати у разі їх використання, наприклад, домашня сторінка, головна сторінка, WWW, інтернет, інтернет-сторінка, web-сторінка.

Практичні поради щодо заповнення *TITLE*–тегу**.** Спочатку потрібно створити список ключових слів для сторінок, які оптимізуються.

Приклад. Розглянемо список ключових слів для однієї зі сторінок сайта квіткових магазинів, розташованих у Києві та Львові:

<TITLE> фітодизайн, замовлення букетів, доставка квітів і букетів по Києву </TITLE>.

Далі варто розмістити ключові слова у порядку зниження їх важливості, тобто на перше місце поставити слова із найвищим рангом (найважливішим змістовним наповненням).

Приклад. Для розглядуваних сторінок сайта квіткових магазинів отримаємо:

- квітковий магазин Київ;
- квітковий магазин Львів;
- весільні букети.

Далі необхідно скласти пропозицію із цих фраз, використовуючи якнайменше слів, тому що кожне додаткове слово знижує релевантну вагу інших.

Приклад. У разі сторінок сайта квіткових магазинів необхідно фразу «Квітковий магазин у Києві та Львові, який спеціалізується на весільних букетах» сформулювати так: «Квітковий магазин Київ Львів спеціалізується на весільних букетах». У цьому разі найбільш важливі слова містяться спочатку тегу, тому немає потреби повторювати словосполучення «квітковий магазин» двічі, тому що отримана пропозиція містить обидві фрази: «квітковий магазин у Києві» та «квітковий магазин у Львові». Більшість пошукових машин проігнорують слово «на» або зреагують як на «стоп-слово».

Слово «Львів» правильно було б виділити комами з точки зору пунктуації, але у багатьох пошукових машин коми є роздільниками ключових слів. Слово «весільні букети» перебуває в самому кінці, що може знизити релевантну вагу цієї фрази, оскільки пошукові машини вважають більш важливими слова на початку заголовка.

Враховуючи розглянуті фактори, заголовок набуває такого вигляду:

Квітковий магазин Київ Львів: весільні букети на будь-який смак.

Двокрапку (або тире) пошукова машина не сприймає, як «стоп-символ».

Отриманий заголовок охоплює такі запити:

- квітковий магазин Київ;
- квітковий магазин Львів;
- квітковий магазин у Києві;
- квітковий магазин у Львові;
- квітковий магазин у Києві та Львові;
- весільні букети;
- Київ весільні букети;
- Львів весільні букети.

Порядок слів важливий: якщо необхідно домогтися максимального рангу за запитом «Львівський квітковий магазин», то «ключі» необхідно розміщувати саме в такому порядку (а не «квітковий магазин Львів»), тому що пошуковий робот шукає точну відповідність запиту.

Якщо власник сайта хоче записати назву компанії в тег «*TITLE»*, то краще її поставити в кінець, щоб важливі ключі одержали більш релевантну вагу. Разом із тим, бувають і винятки, коли назва компанії свідчить про напрям діяльності, наприклад, квітковий магазин названо «Веселий квіткар»:

<TITLE> «Веселий квіткар» – квітковий магазин Київ Львів: весільні букети на любий смак </TITLE>

Це може дещо знизити релевантну вагу інших ключів, зате дозволить установити бренд на головній сторінці, що може бути важливим у цьому разі.

6. Стилістичне оформлення тексту. Пошукові системи надають особливе значення тексту, який певним чином виділено на сторінці. Для удосконалення стилістичного оформлення тексту можна надати такі рекомендації щодо його оформлення:

− бажано використовувати ключові слова в заголовках (текст, виділений тегами «H», особливо «h1» і «h2»), якщо сторінка вже функціонує, то нині можна перевизначити вид тексту, виділеного цими тегами, за допомогою *css*;

− загалом тег <hl> значно важливіший, ніж тег <h2>, який, у свою чергу, важливіший, ніж тег <h3>, і т. д.;

− бажано виділити ключові слова жирним шрифтом (не в усьому тексті, а 2…3 рази на сторінці), використовуючи тег «*strong*», замість більш традиційного тегу «*B*» (bold).

7. Теги «*ALT*» зображень. Будь-яке зображення на сторінці має спеціальний атрибут «*альтернативний текст*», який задається у значенні тегу «*ALT*». Цей текст буде відображений на екрані за умови, що не вдалося завантажити зображення або показ зображень заблоковано у браузері. Пошукові системи запам'ятовують значення тег*а* «*ALT*» у процесі аналізу (індексації) сторінки, проте не використовують його під час ранжування результатів пошуку.

Приклад. У разі створення сторінок сайта квіткових магазинів необхідно використати тег «*ALT*» так:

<IMG SRC = "bouquet-photo.jpg" alt = "Весняний весільний букет" width = 123 height = 52>.

Нині пошукова система Google враховує текст у значенні тегу «*ALT*» тих зображень, які є посиланнями на інші сторінки, решта тегів «*ALT*» ігноруються, з чого можна зробити висновок, що

використовувати ключові слова в тегах «*ALT*» можна і потрібно, хоча принципового значення це не має.

8. Мета-тег «*Desciption*» є невидимим на сторінці текстом, вписаним у HTML-документ, призначений для опису змісту сторінки для пошукової машини. Пошукові роботи збирають цю інформацію у процесі індексування й часто використовують її як короткий опис сайта в лістингах. Цей тег ніяк не впливає на ранжування, але є дуже важливим. Багато пошукових систем (зокрема, Google) відображають інформацію з цього тегу в результатах пошуку, якщо він є на сторінці та його вміст відповідає вмісту сторінки і пошуковому запиту.

Зауважимо, що високе місце в результатах пошуку не завжди забезпечує велику кількість відвідувачів. Якщо опис інших сторінок-конкурентів у результатах представлення посилань користувачеві буде привабливішим, ніж у сайту, то відвідувачі пошукової системи виберуть саме їх, а не цей ресурс.

З огляду на це грамотно складений мета-тег «*Description*» має суттєве значення: опис має бути коротким, але інформативним і привабливим, містити ключові слова, характерні для цієї сторінки. Якщо не заповнювати мета-тег, пошукова машина сама зробить опис сайта, який ґрунтуватиметься на довільному тексті, взятому в будь-якому місці сторінки.

Практичні поради з написання мета-опису такі. Мета-опис може бути достатньо довгим, але тільки певна його частина буде індексуватися й відображатися в лістингах – не більше 200…250 символів, тому бажано, щоб важливі ключові слова згадувалися спочатку.

Приклад. Розглянемо лаконічне формулювання мета-опису. Якщо потрібно знайти квітковий магазин у Києві або Львові, то «Веселий квіткар» пропонує кращі весільні букети, оформлення подарунків і свят на всі випадки життя.

Довжина складеного тексту становить близько 150 символів, куди ввійшлиі основні ключові слова, але бажано додавати ще певні заклики, щоб залучити відвідувача, наприклад:

Зробіть *online*-замовлення з 10-процентною знижкою!

Виходячи з усіх підходів щодо удосконалення мета-опису формуємо такий тег:

<META name = "description" content = Якщо Ви шукаєте квітковий магазин у Києві або Львові, то «Веселий квіткар» пропонує кращі весільні букети, оформлення подарунків і свят на всі випадки життя. Зробіть online-замовлення з 10-процентною знижкою!">

Підготовка структури сайта. Збільшення кількості сторінок сайта покращує його видимість у пошукових системах. Крім того, поступове додавання нових інформаційних матеріалів на сайт сприймають пошукові системи як розвиток сайта, що може надати додаткової у процесі ранжування, тому необхідно розміщувати на сайті більше інформації: новини, прес-релізи, статті, корисні поради.

Структура директорій має суттєве значення. Пошукові машини в середньому індексують на сайті не більше 50…60 файлів, причому тільки до другого рівня вкладеності, тому важливу інформацію слід зберігати не глибше другої вкладеної директорії, наприклад: *www.site.ru/levell/level2/page.htm.,* а **у** URL має бути якнайменше папок, тому що сторінки, які містяться під декількома рівнями, матимуть меншу вагу для пошукових систем, будуть швидко індексовані та не отримають високих рейтингів.

Для організації структури сайта зазвичай застосовують директорії, але у директоріях, які містять лише два або три рівні, важко організувати логічну структуру, тому якщо для створення структури сайта потрібно мати директорії з більшою кількістю рівнів, то необхідно використовувати карту сайта.

Разом з тим, якщо вже встановлено певну структуру директорій, і сторінки посідають високі місця в рейтингу, то не треба змінювати

структуру, інакше через незнання нової адреси web-сторінки пошукові машини можуть видалити такі сторінки з бази пошуку.

Каскадні таблиці стилів (Cascading Style Sheets, CSS) – простий інструментарій створення стилів (шрифт, кольори тощо) web-документів, за якого весь код, що задає стилі (всю інформацію про те, як сторінка має виглядати), зберігається в окремому файлі, що зменшує обсяг коду на кожній сторінці, а також обсяг коду, який пошуковій машині потрібно проіндексувати. Чим менше коду пошукова машина переглядає, тим швидше індексується й відображається важливий вміст і тим більше сторінок може бути збережено в базі пошукової машини.

За допомогою CSS можна зробити так, щоб текст, наповнений ключовими словами, з'явився на початку вихідного тексту HTML, незалежно від того, де цей текст з'являється на видимій частині web-сторінки. Але у разі частого виростання CSS слід перевіряти сайт, змінюючи розміри шрифтів у браузері, щоб переконатися, що інформація коректно відображатиметься за будь-яких умов.

Меню, які розкриваються. Більшість меню, які розкриваються, несумісні з пошуковими машинами, тому що вимагають виконання сценаріїв на стороні клієнта або сервера. Пошукові машини не можуть виконувати ці сценарії або мишею розкривати меню, як це роблять відвідувачі, тому пошуковий робот не має доступу до посилань.

Недоліки графічних навігаційних елементів, таких як кнопки, що розкривають меню й карти-зображення, такі:

- пошукові машини мають застосовувати Alt IMG-теги, щоб зрозуміти їх;
- пошукові машини не можуть виконувати JavaScript- або CGI-сценаріїв;
- не всі пошукові машини можуть переходити за графічними посиланнями;
- графічні посилання завантажуються довше, ніж текстові.

Одна з ефективних схем навігації – використання кнопок (зі сценаріями JavaScript або без них) і дублювання меню відповідними текстовими посиланнями знизу сторінки. У цьому разі можна вставляти ключові слова у якірний текст посилань і в атрибут Alt графічних кнопок.

Зовнішня оптимізація. Зовнішні посилання розміщуються на певних ресурсах на вільній або платній основі, зокрема в каталогах сайтів (таке розміщення не рекомендують для нових ресурсів), на сайтах схожої тематики та регіону (останнім часом актуально), в каталогах статей та прес-релізів і багатьох інших сайтів.

У разі купівлі посилань для підвищення ваги посилань сторінки необхідно пройти такі етапи:

1) визначення бюджету на посилання;
2) складання текстів посилань;
3) розміщення посилань на зовнішніх ресурсах;
4) контроль розміщених посилань;
5) аналіз ефективності розміщення;
6) корегування стратегії;
7) підготовка звіту;
8) оптимізація і корегування бюджету.

Нормативний текст. Важливе значення під час ранжування результатів пошуку надають тексту зовнішніх посилань на сайт.

Текстом посилання (або якірним) називають текст, який міститься між тегами «*A*» і «*/A*» і по якому можна клацнути курсором миші у браузері для переходу на нову сторінку. Якщо текст посилання містить потрібні ключові слова, то пошукова система сприймає їх як додаткову і дуже важливу рекомендацію, підтвердження того, що сайт дійсно містить цінну інформацію, яка відповідає темі пошукового запиту.

Релевантність посилань. У разі ранжування сайту пошукові машини порівнюють також інформаційний вміст сторінок, які пов’язані

посиланнями, тобто їх релівантність (відповідність одному змістовному наповненню).

Приклад. Припустімо, просувають ресурс із продажу автомобілів. У цьому разі посилання із сайта по ремонту автомобілів означатиме значно більше, ніж аналогічне посилання із сайта із садівництва. Перше посилання відповідає тематично схожому ресурсу, тому буде більшою мірою оцінене пошуковою системою.

Соціальна інтеграція web-сайта. Багато сучасних компаній використовують соціальні мережі для просування свого контенту в мережі, комунікації зі споживачами, а також для проведення соціологічних опитувань і досліджень. Соціальні мережі мають значний потенціал у просуванні web-сайта, а також позитивний вплив на рейтинг сайта має інтеграція із сервісами блогінгу (livejournal) і мікроблогінгу (Twitter).

Реєстрація сайту в каталогах. Перед реєстрацією в пошукових машинах необхідно переконатися, що сайт готовий до індексування, тобто:

– Чи готові всі сторінки? (Немає сторінок «under construction»).

– Чи всі посилання дійсні? (Немає повислих посилань).

– Чи всі сторінки оптимізовані?

– Чи всі сторінки сумісні з пошуковими машинами?

– Чи використано Robots.txt або Мета-тег для пошукової роботи, щоб позначити неіндексовані сторінки, (наприклад, купівельний кошик)?

– Якщо сайт призначений для електронної комерції, то чи протестовано процес замовлення товарів і купівельні кошики?

– Чи протестовано сайт на usability (зручність користування сайтом)?

Для спрощення процесу реєстрації потрібноно приготувати текстовий файл або таблицю з такою інформацією:

1. URL головної сторінки.
2. URL інших сторінок, які треба зареєструвати.
3. Назва сайта, яка видаватиметься в пошукових лістингах (зазвичай це назва компанії).
4. Короткий опис сайта (10…20 слів).
5. Докладний опис сайта (30…50 слів).
6. Список ключових слів.
7. Ім'я людини, яка реєструє сайт у каталозі.
8. E-mail (необхідно вказувати діючий e-mail, який регулярно перевіряється, але в якому є фільтри спама, тому що в результаті реєстрації обов'язково надходитиме спам).
9. Адреса й контактна інформація компанії (для деяких каталогів).

Бажано пам'ятати e-mail, вказаний під час реєстрації, оскільки деякі каталоги вимагають посилатися на нього, якщо треба змінити інформацію про сайт. Для збору інформації про сайт можна використовувати текстовий файл блокнота, хоча можна застосовувати й документ Word, а також таблицю.

Варто якнайчастіше вживати ключові слова в описі сайта, а також використати оптимізовані Title- і Мета-теги, якщо текст не перенасичений ключовими словами й ці теги не стали спамоподібними.

Правила реєстрації такі:

1. Реєстрація здійснюється лише один раз: ніколи не буває потреби в повторній реєстрації, за винятком досить рідких випадків, коли сайт не бачить пошукова машина.
2. Правильність - пошук розділу каталогу, який найбільш підходить для сайта.
3. Стислість - опис сайта надається в одній-двох пропозиціях.
4. Точність - не варто обманювати потенційних відвідувачів за допомогою неправильного опису.

5. Релевантність – співвідношення між релевантністю реєстраційного опису й оптимальною кількістю ключових слів, слід не завищувати це співвідношення, переповнивши опис ключовими словами.

6. Скромність - заява «Кращий сайт у світі!» нікого не переконає, але може обурити редакторів каталогу.

7. Терплячість - індексація сайта може тривати до шести місяців, повторна реєстрація не вирішить проблеми, а може тільки перемістити в кінець черги.

Каталог DMOZ (www.dmoz.org), або Open Directory Project, - найбільший каталог Internet, окрім якого в Internet є велика кількість копій основного сайта DMOZ. Розмістивши сайт у каталозі DMOZ, можно отримати не лише цінне посилання з самого каталогу, але й ще кілька десятків посилань від споріднених йому ресурсів, тому каталог DMOZ має суттєву цінність для web-мастера.

Для успішної реєстрації в DMOZ необхідно вжити таких заходів:

1. Заповнити реєстраційну заяву.
2. Чекати три місяці.
3. Надіслати листа редактору розділу.
4. Чекати три місяці.
5. Відправити листа редактору розділу.
6. Чекати три місяці.
7. Просити допомоги на форумі Open Directory Public Forum (*http://resource-zone.com*).
8. Чекати один місяць.
9. Надіслати листа головному редактору DMOZ, просити допомоги на різних форумах.

Підтримка сайта. На цьому етапі здійснюють аналіз досягнутих результатів та подальше коригування сайта, а також бажано розмістити

рекламу, враховуючи аудиторію сайтів, на яких буде розміщено рекламу.

Приклад. Студія Артемія Лебедєва на етапі свого створення організувала премію на кращий сайт, якою було нагороджено всі найбільші сайти разом із банером, на якому був напис «Лауреат премії - кращий сайт», який був посиланням на сайт студії. Як результат - величезна кількість посилань на сайт зі сторінок із високим PageRank.

Створення й ведення блогів. Блог (похідне слово від «Web log») – це сторінка, яку створює або окрема особа, або організація, на яку користувачі можуть додавати свої коментарі й ідеї з обговорюваної теми.

Блоги відомих експертів є надійним інформаційним джерелом з багатьох тем для великої кількості користувачів, їм досить просто досягти високих позицій у рейтингах пошукових машин, а також вони стали майданчиком для спамерів, але лише доти, поки Google не перестав індексувати їх коментарі.

Блоги можна використовувати в різних цілях, наприклад, з їх допомогою друзі, які перебувають на далеких відстанях, можуть листуватися між собою, або співробітники фірм - обговорювати робочі питання, перебуваючи в різних офісах.

Зазвичай блоги відкривають для обговорення якої-небудь конкретної тематики, зокрема футболу, політики, книг тощо. Використовуючи блог достатньо написати повідомлення (статтю, замітку, новину) і воно публікується зверху блогу, над всіма іншими повідомленнями.

Пошукові системи індексують блоги, завдяки чому можна використати їх ведення як інструмент для просування web-ресурсів, оскільки вони мають такі властивості:

- унікальний контент блогів;
- постійно оновлюваний контент;
- простоту дизайну.

Для просування web-ресурсів можна створити блог і розміщувати на ньому інформацію про нові продукти, послуги, статті співробітників тощо.

Тематичні форуми. Форум – це такий вид сайта, де відвідувачі можуть висловлюватися з певної теми на спеціально відведеній для неї сторінці.

Як і блоги, форуми можуть бути джерелом релевантної інформації, оскільки ціль деяких користувачів – лише залишити посилання на свій сайт, що прийнятно, якщо посилання релевантне й може допомогти іншим членам форуму. Проте через велику кількість нерелевантних спамерських посилань на форумах такі посилання ставатимуть неефективними, оскільки деякі власники форумів створюють їх лише для підвищення рейтингів.

Для досягнення суттєвих результатів необхідно звернутися до копірайтера із проханням представляти на Internet-форумах особу компанії, яка просувається, що дасть можливість прямо поспілкуватися із клієнтами. Форум, додаючи вхідні посилання, допомагає розмістити їх на свої ресурси, що полегшує користувачам знаходження потрібного сайта. Щоб форум міг бути ефективним ресурсом, потрібно отримати довіру в його користувачів.

Поширення прес-релізів. Прес-релізи – ще один спосіб залучити відвідувачів на сайт, оскільки новини поширюються швидко за допомогою Google News або інших систем, а також прес-релізи часто архівуватимуться у значній кількості місць і залишатимуться активними протягом багатьох років, забезпечуючи надійні посилання для підвищення позицій у рейтингах.

Бажано самостійно зібрати базу даних сайтів, які публікують прес-релізи з тематики бізнесу, або придбати вже готову базу із загальної тематики.

Поширення статей. Статті – це ефективний спосіб залучити вхідні посилання, ефективність від використання статті зростає, коли її розміщують там, де є посилання на сайт, який просувається на інших сайтах. Розміщення такого контента на сайті – не лише спосіб додати вхідних посилань, але й отримати перевагу від розміщення статей на багатьох сайтах для одержання зворотних посилань.

Каталоги статей – один з надійних способів одержання вхідних посилань, які не вимагають суттєвої професійної підготовки, знання ринку бізнесу й розвитку відносин з партнерами, які можуть розмістити ці статті на своїх ресурсах. Розміщення статей на авторитетних сайтах підвищує рейтинг, але варто отримати багато посилань у короткий термін для швидкого його підвищення, використовуючи каталоги статей, яких може бути сотні, але раціональною є кількість каталогів не більше десяти. Якщо всі посилання сайта, який просувається, надходитимуть до нього винятково від каталогів статей, то у пошукових машин це може викликати підозру, тому бажано, щоб каталоги статей залучили 20% посилань сайта, який просувається.

Статті, які додають у каталоги, мають бути не занадто довгими (близько 600 слів), відформатованими так, щоб було зручно читати, бажано інформацію розміщувати за можливістю у списках, тому що це ділить її на дрібні шматочки, які легко сприймаються, мають мати змістовні заголовки та містити цікавий матеріал за актуальними темами. Найбільш прийнятні заголовки можуть бути такими: «10 кроків до...» або «10 способів, які...», або «сім типових помилок, які ....».

Проблеми дублювання матеріалів. Поширення статей є способом збільшення кількості вхідних посилань, однак воно може призвести й до їх зменшення, якщо розмістити одну статтю одночасно на декількох сайтах, які мають суттєвий рейтинг, оскільки у разі, якщо

пошукові машини знаходять статті з однаковим змістом, то вони показують лише одну з них.

Неможливо контролювати, яке джерело статті машина вважатиме за первісне: зазвичай це сторінка, де стаття була вперше проіндексована, але іноді це сторінка з більшою кількістю вхідних посилань.

Здебільшого пошукові машини не показують у результатах всі сторінки, де є стаття, але негативний вплив статей з дублюючим змістом має суттєве значення під час визначення рейтингу, тому варто дозволяти іншим сайтам публікувати лише уривок зі статті сайта, який просувається, щоб пошукові машини бачили унікальність цього сайта й виводили його в результати першими. Крім того, це зробить сайт більш рейтинговим в очах відвідувачів, оскільки на ньому подано повну версію матеріалів.

Публікуючи статтю на сайті слід переконатись у врахуванні таких правил:

- дотримання авторського права;
- наявність посилання на сайт автора;
- здійснити пошук ключових фраз з матеріалів сайту, який просувається, за допомогою пошукових машин, щоб запобігти несанкціонованому використанню статей, для чого можна використати Google Alerts (http: //www. google.com/alerts).

8.6.3. Оптимізація PageRank

PageRank - найважчий для маніпулювання фактор під час удосконалення рейтингу сторінок, який складно як визначити, так і утримувати його високе значення.

Є три основні основні підходи щодо оптимізації PageRank, а саме такі:

– визначення сторінок, які обирати для отримання посилань на сайт, а також працевитрати на це;

– визначення сайтів, на які розміщувати посилання зі свого сайта, та на якій сторінці сайта ставити їх посилання;

– удосконалення внутрішньої навігаційної структури і зв'язків сторінок для створення максимального зворотного зв'язку PageRank.

Посилання із сайта. Щоб визначити найкращу стратегію проставляння посилань із сайта, спочатку потрібно розглянути посилання на сайт, тобто припустити, що наявні посилання, які вказують на сайт із каталогів, подібних DMOZ і Yahoo, які дають йому невелике прирощення PageRank. Використовуючи внутрішні сторінки сайта, можна керувати зворотним зв'язком значно краще, ніж за допомогою посилань на зовнішні сторінки.

У загальному випадку потрібно зберегти PageRank усередині сайта, тобто посилатися на зовнішні сторінки лише зі сторінки сайта, яка має низький PageRank і містить велику кількість внутрішніх посилань, які вказують на інші сторінки сайта.

Отже, розміщуючи зовнішнє посилання, необхідно віддавати перевагу тим сторінкам, які або посилаються на ту сторінку сайта, яка міститься сторінкою вище від сторінки, з якої посилаються (наприклад, зовнішня сторінка A посилається на сторінку B_1, яка, у свою чергу, посилається на сторінку B_2, на якій і розміщено посилання на зовнішню A), або сторінкам, які посилаються на ту сторінку, яка посилається на сторінку, що посилається на посилальну сторінку ($A \rightarrow B_1$, $B_1 \rightarrow B_2$, $B_2 \rightarrow B_3$, $B_3 \rightarrow A$). Можливо отримати суттєве підвищення PageRank, якщо посилання із зовнішніх сайтів не вказують на сторінку з посиланнями.

Одним зі способів організації таких посиланнь буде написання оглядів сайтів, на які виконується посилання на окремій сторінці сайта, і забезпечення посилання на ці огляди разом із кожним посиланням на

зовнішній сайт. Необов'язково, але бажано, щоб ці сторінки відкривали в іншому вікні, але не за допомогою JavaScript, тому що роботи пошукових систем не можуть прямувати java-посиланнями.

Для підвищення значення PageRank слід первірити, щоб сторінка оглядів посилалася на сторінку, яка міститься вище у структурі сайта (наприклад, головна або будь-яка інша важлива сторінка), щоб зменшити значення ваги PageRank, яке переходить із сайту, і забезпечити підвищення більшої частина ваги PageRank, яка залишається, за рахунок ефекту зворотного зв'язку. Спрямувавши цей зворотний зв'язок на головну сторінку, можна гарантувати, що менше значення ваги передається назад сторінці з посиланням і більше залишається на сайті. На сторінці з посиланням також потрібно поставити посилання на головну сторінку та інші значущі сторінки сайта. Однак не можна розміщувати інші посилання на сторінці з оглядом (окрім посилання на головну сторінку).

Внутрішня структура та зв'язки. Чим більше сторінок конкретний сайт має в індексі Google, тим вище значення його початкової сумарної ваги PageRank, а також обчисленої ваги PageRank, з якою пошукова машина Google має працювати. Оскільки кожній сторінці задано однакове початкове значення PageRank до того, як починає обчислюватися PageRank, то більша кількість сторінок здатна його підвищити. Отже, якщо спочатку сайт містить більшу кількість сторінок, то й ефект зворотного зв'язку буде вищим.

Зворотний зв'язок – це природний процес для PageRank, який проходить у внутрішніх посиланнях сайта і є критичним, щоб Google зміг оцінити, які сторінки сайта важливі. Для процесу визначення кількості посилань зворотного зв'язку характерно, якщо сайт не матиме вхідних чи вихідних посилань (посилань із зовнішніх сайтів і на зовнішні сайти відповідно), то структура сайта забезпечить таку саму кількість посилань зворотнього зв'язку.

8.7. Висновки

1. Для підвищення позиції сторінки в результатах пошуку за запитом необхідно розміщувати зовнішні й внутрішні посилання на цю сторінку. Чим більше посилань, чим більш авторитетні сторінки, на яких розміщені посилання, - тим краща позиція.

2. Для того, щоб досягти високих позицій у результатах пошуку пошукової машини за певним запитом, потрібно дотримуватися певних правил, розробляючи структуру сайта та заповняючи змістову частину.

3. Позицію також підвищує врахування фактора ранжування, тобто розміщення в тексті посилання тексту запиту, за яким просувається сторінка.

4. Оцінити авторитетність сторінки можна за непрямими показниками, зокрема PageRank.

5. Обов'язково необхідно перевірити, чи дійсно інформація на сторінках відповідає запитам, за якими планується просування цієї сторінки.

6. Проаналізувавши всі рекомендації, можна скласти загальний план оптимізації web-сайта.

7. Бажано скласти наближений список ключових слів і перевірити рівень конкуренції за ними, за найбільш цікавими запитами виконати ґрунтовний аналіз видачі пошукових систем, за результатами якого ухвалити остаточні рішення щодо ключових слів.

8. Наповнити web-сайт якісним текстом. За наявності якісного інформаційного вмісту легше отримувати зовнішні посилання та відвідувачів.

9. Зареєструвати сайт в каталогах автоматично або вручну.

10. Перевірити індексацію (підтвердження того, що сайт нормально сприймають різні пошукові системи).

11. Здійснити моніторинг позицій сайта за потрібними ключовими словами.

12. Постійно намагатися підвищувати популярність сайта за посиланнями.

13. Проводити аналіз відвідуваності й тенденцій користувачів за допомогою модуля статистики, наприклад Google Analytics.

8.8. Запитання для самоконтролю

1. Які слова краще не використовувати у *TITLE*-тегу?
2. Чи має значення порядок слів у *TITLE*-тегу?
3. Навіщо треба заповнювати *META*-тег?
4. Як краще розміщувати текст на сторінці?
5. У яких випадках слід ураховувати близькість ключових слів один до одного?
6. Якою має бути структура директорій?
7. Якими мають бути текстові посилання?
8. Наскільки важливий атрибут *Alt* тегу *IMG*?
9. Чи варто використовувати каскадні таблиці стилів (CSS) та чому?
10. Що таке PageRank?
11. За яким принципом розраховують PageRank?
12. Що таке зворотний зв'язок PageRank?
13. Яким чином краще організувати структуру внутрішніх посилань?
14. Чи варто створювати багато сторінок на одному сайті?
15. Які типи посилань розрізняють?
16. Які виокремлюють способи отримання посилань? Назвіть найбільш ефективні нині.
17. Чи ефективні посилання з каталогів посилань?
18. На чому грунтується метод оптимізації з використанням статей?
19. Який каталог посилань важливий для Google?

20. Який основний параметр якості посилання?

21. Як відбувається реєстрація сайта в каталогах? Назвіть правила реєстрації.

22. Що таке блог? Вкажіть його призначення.

23. В чому полягають переваги блогів для пошукових систем?

24. Для чого потрібні статті? Опишіть методи їх поширення.

25. Для чого призначені тематичні форуми?

26. Як визначити правильний баланс посилань?

27. Що треба враховувати, оцінюючи якість посилань?

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. **Van Steen M.,** Homburg P., Tanenbaum A.: «Globe: A Wide-Area Distributed System», IEEE Concurrency, vol. 7, №1, pp. 70-78, Jan. 1999.

5. **Homburg P.C.** The Architecture of a Worldwide System. Ph.D Thesis, Vrije University Amsterdam, Department of Mathematics and Computer Science, 2001.

6. **С. Морган.** Разработка распределенных приложений на платформе Microsoft .Net Framework/ С. Морган, Б. Райан, Ш. Хорн, М. Бломсма,
2008 – 608 с.

7. **Wang H.,** Consumer Privacy Concerns about Internet Marketing/ Wang H., LO, M. K., Wang C. – Commun : ACM, 1998.

8. **Ноутон П.,** Java 2/ Ноутон П., Шилдт Г. – Петербург: БХВ, 2001.-1102 с. -ISBN: 978-5-94157-012-6, 0-07-211976-4.

9. **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.

10. **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

11. **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.

12. **Bhoedjang R.** User-Level Network Interface Protocols// Bhoedjang R., Ruhl T., Bal H. IEEE Computer, 1998. – pp. 53 – 60.

13. **Day J.** The OSI Reference Model.// Day J., Zimmerman H. Proceedings of the IEEE, Dec. 1983.

14. **Comer D**. Internetworking with TCP/IP// Comer D., Volume I: Principles, Protocols, and Architecture. Upper Saddle River, NJ: Prentice Hall, 4th ed., 2000.

15. **Handel R.** ATM Networks// Handel R., Huber M., Schroder S. Wokingham, Addison- Wesley, 2nd ed., 1994.

16. **Schulzrinne, H.** RTP: A Transport Protocol for Real-Time AppUcations/ Schulzrinne, H., Casner S., Frederick R., Jacobson V. - RFC 1889, Jan. 1996 – p.104

17. **Lewis B.** Multithreaded Programming with Pthreads. Englewood Cliffs/ Lewis B., Berg D.J. NJ: Prentice Hall, 2nd ed., 1998.

18. **Stevens W.** UNIX Network Programming – Interprocess Communication/ Stevens W. – Prentice Hall Ptr, 1998 – 556 с.

19. **Triantafillou P.** Achieving Strong Consistency in a Distributed File System// Triantafillou P., Neilson C. - IEEE Trans. Softw. Eng., Jan. 1997 – pp. 35-55.

20. **Katz E.** A Scalable HTTP Server: The NCSA Prototype// Katz E., Butler M., McGrath R. - Сотр. Netw. & ISDN Syst., Sept. 1994 – p.10.

21. **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.

22. **Lewis B.** Multithreaded Programming with Pthreads// Lewis B., Berg D.J. – Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1998.

23. **А. Цимбал** Технологии создания распределенных систем. Для профессионалов/ А. Цимбал, М. Аншина – Питер, 2003. – 576 с.

24. **Stevens W.** UNIX Network Programming – Interprocess Communication/ Stevens W., 1999 – p. 558.

25. **Stevens W.** Advanced Prograinming in the UNIX Environment/ Stevens W., Reading, MA: Addison-Wesley, 1992 – p. 960.

26. **Stevens W.** UNIX Network Programming – Networking APIs/ Stevens W., Sockets and XTI. Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1998.

27. **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.

28. **Radia S.** Names, Contexts, and Closure Mechanisms in Distributed Computing Environments// Radia S., Ph.D. Thesis, University of Waterloo, Ontario, 1989.

29. **Rao H.** Accessing Files in an Internet: The Jade File System// Rao H. and Peterson L., IEEE Trans. Softw. Eng., June 1993- p.29.

30. **Lampson B.** Designing a Global Name Service// Lampson B., Proc. Fourth Symp. on Principles of Distributed Computing. ACM, 1986 – p.10.

31. **Vixie P.** A DNS RR for Specifying the Location of Services (DNS SRV)// Vixie P., RFC 2052, Oct. 1996.

32. **Katz E.** A Scalable HTTP Server: The NCSA Prototype// Katz E., Butler M., and Mcgrath R., Сотр. Netw. & ISDN Syst. 1994.

33. **Andrews G.** Foundations of Multithreaded, Parallel, and Distributed
Programming/ Andrews G., Reading, MA: Addison-Wesley, 2000.

34. **Singhal M.** Advanced Concepts in Operating Systems/ Singhal M. and Shivaratri N.: Distributed, Database, and Multiprocessor Operating Systems. New York: McGraw- Hill, 1994.

35. **Wu J.** Distributed System Design. Boca Raton// Wu J., CRC Press, 1998.

36. **Drummond R.** Low-Cost Clock Synchronization// Drummond R. and Babaoglu O., Distributed Computing, 1993 – p. 10.

37. **Б. Эккель** Философия Java. Библиотека программиста/ Б. Эккель, 4-е изд. 2009 – 638 с.

38. **B. Chapman** Extending HPF for advanced data-parallel applications//
B. Chapman, P. Mehrotra, H. Zima, IEEE Parallel and Distributed Technology, 1994 – pp.15-27.

39. **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.

40. **D.Y. Cheng** A survey of parallel programming languages and tools// D.Y. Cheng, Moffett Field, 1993.

41. **Ільченко М.Ю.** Сучасні телекомунікаційні системи / Ільченко М.Ю., Кравчук С.О. – НВП "Видавництво "Наукова думка" НАН України", 2008. – 328 с.

42. **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.

43. **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.

44. **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.

45. **Романов А.И.** Телекоммуникационные сети и управление / Романов А.И. - Киев, ВПЦ «Киевский Университет», 2003. - 247с.

46. **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.

47. **Кравчук С.О.** Основы программирования на языке Java: Учеб. пособ. / Кравчук С.О., Шонін В.О., – К.: Норита плюс, 2007. – 280 с.

48. **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.

49. **Наритник Т.М.** Радіорелейні та тропосферні системи передачі: навчальний посібник / Наритник Т.М., Волков В.В. - ІВЦ "Політехніка", 2009р. -331с.

50. Електронное руководство FreeBSD handbook. (http://www.freebsd.org/handbook/index.html).

51. **Rob Ross** Beowulf HOWTO/ Rob Ross (http://www.linuxdoc.org/HOWTO/Beowulf-HOWTO.html).

52. **Gabriele Kotsis** Interconnection Topologies and Routing for Parallel Processing Systems/ Gabriele Kotsis http://www.ani.univie.ac.at/~gabi/papers/in.ps.gz.

53. **Ian Foster** Designing and Building Parallel Programs/ Ian Foster, Addison-Wesley, 1995.

54. **Jonathan M.D.** Hill. An introduction to the data-parallel paradigm/ Jonathan M.D., Department of Computer Science, 1994.

55. **K. Hwang** Advanced Computer Architecture: Parallelism, Scalability, Programmability/ K. Hwang, McGRAW-HILL, 1993.

56. **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.

57. **Kai Hwang** Computer Architecture and Parallel Processing/ Kai Hwang, Faye A. Briggs, McGRAW-HILL, 1986.

58. **Mark Baker Cluster** Computing White Paper/ Mark Baker, 2000 – p.119.

59. **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.

60. **P. Mehrotra** Programming distributed memory architectures. In Advances in Languages and Compilers for Parallel Computing/ P. Mehrotra and J. Van Rosendale, MIT Press, 1991.

61. **Rajkumar Buyya** High Performance Cluster Computing: Architectures and Systems/ Rajkumar Buyya, Prentice Hall PTR, 1999 – p.849.

62. **V. Kumar** Scalable load balancing techniques for parallel computers//

V. Kumar, A. Grama, and V. Rao, Parallel and Distributed Computing, 1994 – pp. 60-79.

63. **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.

64. **Graham E. Fagg** PVMPI: An integration of the PVM and MPI systems// Graham E. Fagg, Jack J. Dongarra, Calculateurs Paralleles, 1996 – pp. 595-605.

65. **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.

66. **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)

67. **Hans Bergsten** An Introduction to Java Servlets/ Hans Bergsten (http://www.webdevelopersjournal.com/articles/intro to servlets.html)

68. **Хейфец И.** Архитектура .NET/ Хейфец И. (обзор) (http://www.gotdotnet.ru/default.aspx?s=doc&d no=24&c по=4).

69. **Филев А.** Сравнивая .NET и Java// Филев А., 2010. (http://www.gotdotnet.ru/blogs/andrew-filev/6383/).

70. **Арчер Т.** Основы C#. Новейшие технологии/ Арчер Т. - М.: Издательско-торговый дом «Русская редакция», 2001 – 448 с.

71. **Старостин Д.** Новый «универсальный клей» - Web Services. Microsoft// Старостин Д., 2005. (http://esco-ecosys.narod.ru/2005_1/art13.htm).

72. **Рейли Д.** Создание приложений Microsoft ASP.NET/ Рейли Д. - М.: Издательско-торговый дом «Русская редакция», 2002 – 480 с.

73. Network Protocols Handbook. – Javvin Technologies, 2005 -340 с.

74. Спецификации SOAP 1.2 (http://www.w3.org/TR/soap/).

75. **Ньюкомер Э.** Веб-сервисы. XML, WSDL, SOAP и UDDI/ Ньюкомер Э.–СПб.: Питер, 2003 – 256 с.

76. **Saint-Andre P.** XMPP. The Definitive Guide// Saint-Andre P., Smitb K., Troncon R. –Sebastopol: O'Relly, 2009.

77. **Bogdanov A.** Unified Memory Space Protocol Specification/ Bogdanov A., RFC 3018, December 2000.

78. **Олифер Н.А.** Сетевые операционные системы: учебник для ВУЗов/ Олифер Н.А., Олифер В.Г., – Питер, 2009 – 672 с.

79. Офіційний сайт Gigabyte (http://www.gigabyte.ru).

80. Краш-тест відеокарт (http://comprad.narod.ru/Hardtrable/videocmp.html).

81. **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).

82. **John Sharp.** Microsoft Windows Communication Foundation Step by Step/ John Sharp, – Microsoft Press 2007- 448c., ISBN:9780735623361

83. **Craig McMurtry**. Microsoft Windows Communication Foundation: Hands-on/ Craig McMurtry, Marc Mercuri, – Sams, 2006.

84. **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.

85. **Justin Smith.** Inside Windows Communication Foundation / Justin Smith, – N-Y.: Microsoft Press, 2006, ISBN 9780735623064.

86. **Федоров А.** Windows Azure. Облачная платформа Microsoft/ Федоров А., Мартынов Д., – N-Y.: Microsoft Press, 2010, 100 с.

87. **Фаулер**. Архитектура корпоративных программных приложений/ Фаулер, Мартин, — М.: Издательский дом "Вильяме", 2006. — 544 с.

88. **Я. Фостер.** Анатомия ГРИД. Создание Масштабируемых виртуальных организаций/ **Я. Фостер**, К. Кессельман, С. Тьюк, - 2003.

89. **Я. Фостер**. Физиология ГРИД. Открытая архитектура грид-служб для интеграции распределённых систем/ Я. Фостер, К. Кессельман, Д Ник, С. Тьюк, - 2003.

90. Apache Hadoop. [Електроний ресурс] http://hadoop.apache.org/

91. The Hadoop Distributed File System: Architecture and Design. [Електроний ресурс] http://hadoop.apache.org/core/docs/current/hdfs_design.html

92. Microsoft Dryad. [Електроний ресурс] http://research.microsoft.com/research/sv/dryad/

93. **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.

94. **Anthony Velte**. Cloud Computing: a practical approach. / Anthony Velte, Toby Velte, Robert Elsenpeter, 2010.

95. Amazon Elastic Compute Cloud [Електронний ресурс] http://aws.amazon.com/ec2/

96. Amazon Simple Storage Service [Електронний ресурс] http://aws.amazon.com/s3/

97. What Is Google App Engine? [Електронний ресурс] http://code.google.com/appengine/docs/whatisgoogleappengine.html

98. Google App Engine Developer's Guide: Python [Електронний ресурс] http://code.google.com/appengine/docs/python/overview.html

99. Google Apps [Електронний ресурс] www.google.com/apps/

Eucalyptus documentation [Електронний ресурс]

100. http://open.eucalyptus.com/wiki/Documentation

101. **A. Avizienis.** Basic Concepts and Taxonomy of Dependable and Secure Computing / A. Avizienis, J.-C. Lapri, B. Randell et al. // IEEE Trans. On Dependable Secure Computing. – 2004.– Vol. 1, N 1. – P. 11 – 33.

102. **Харченко В.С.** Гарантоздатні системи та багатоверсійні обчислення: аспекти еволюції // Радіоелектронні і комп'ютерні системи. – 2009. – № 7 (41). – С. 46 – 59.

103. Закон України «Про електронний цифровий підпис» (Відомості Верховної Ради України (ВВР), 2003, N 36, ст.276) {Із змінами, внесеними згідно із Законом N 879-VI (879-17) від 15.01.2009, ВВР, 2009, N 24, ст.296}

104. Материалы по SOA на портале 12NEWS (*http://12news.ru/soa.html)*

105. Oracle SOA (*http://www.oracle.com/us/technologies/soa/)*

106. **SeyedMasoud Sadjadi** – Introduction to CORBA - Software Engineering and Networking Systems Laboratory Department of Computer Science and Engineering Michigan State University (*www.cse.msu.edu/sens).*

107. **Wollrath, Ann**; Riggs, Roger; Waldo, Jim . A Distributed Object Model for the Java System (*http://pdos.csail.mit.edu/6.824/papers/waldo-rmi.pdf).*

108. **Георгиевский А.**, Барышков Д. Распределенная файловая система // GeoКластер. – 2007.

109. Системы электронного управления документами: обзор, классификация и оценка возврата от внедрения. – (*http://www.mdi.ru/library/analit/sysel.html*)

110. **Пахчанян А**. Обзор систем электронного документооборота // Директор информационной службы. – 2001. – № 2. – С. 38 – 42

111. **Göran Husman**, ChristianStåhl. Beginning SharePoint 2010 Administration: Microsoft SharePoint Foundation 2010 and Microsoft SharePoint Server 2010. –Wiley Publishing, Inc., Indianapolis, Indiana

112. Учебник по оптимизации сайта. Практические советы и рекомендации (*http://tutorial.semonitor.ru/#11)*

113. SEO-продвижение: как использовать социальные сети (*http://seodelo.com/)*

114. **И. Ашманов**, А. Иванов. Оптимизация и продвижение сайтов в поисковых системах – Питер, 2008

115. www.sun.com/software/customers/

116. http://www2.sys-con.com/

117. *http://www.labir.ru/article/replica.html*