В. В. Абрамова, С. К. Абрамов

РОЗПОДІЛЕНІ СЕРВІСНІ СИСТЕМИ

2020

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

В. В. Абрамова, С. К. Абрамов

РОЗПОДІЛЕНІ СЕРВІСНІ СИСТЕМИ

Навчальний посібник

Харків «ХАІ» 2020

УДК 004.75(075.8)
А16

Рецензенти: д-р техн. наук, проф. Р. Е. Пащенко,
канд. техн. наук, доц. С. А. Кривенко

Абрамова, В. В.

А16 Розподілені сервісні системи [Електронний ресурс] : навч. посіб. / В. В. Абрамова, С. К. Абрамов. – Харків : Нац. аерокосм. ун-т ім. М. Є. Жуковського «Харків. авіац. ін-т», 2020. – 109 с.

Розглянуто основні поняття й термінологію розподілених обчислювальних систем, а також етапи їх розвитку. Здійснено огляд і порівняльний аналіз архітектур побудови інформаційних систем. Викладено основні принципи організації розподілених обчислень з використанням серверів додатків. Наведено опис загального механізму виклику віддалених процедур, а також приклади його реалізації на базі різних технологій (об'єктні й компонентні розподілені системи, технологія веб-сервісів). Проведено огляд особливостей сервіс-орієнтованого підходу до організації розподілених обчислень. Описано концепцію грід-обчислень і наведено приклади систем, що її реалізують. Висвітлено технологію й принципи організації хмарних обчислень.

Для студентів спеціальності «Телекомунікації та радіотехніка» при вивченні курсу «Розподілені сервісні системи».

Іл. 49. Табл. 3. Бібліогр.: 40 назв **УДК 004.75(075.8)**

1. ЗАГАЛЬНІ ВІДОМОСТІ ПРО РОЗПОДІЛЕНІ СЕРВІСНІ СИСТЕМИ

Сучасний етап еволюції комп'ютерних технологій характеризується бурхливими темпами зростання й розвитку розподілених систем. Формального означення розподіленої системи (РС) сьогодні не існує. Серед безлічі різних означень можна виділити іронічне формулювання першого лауреата премії Дейкстри за досягнення в області розподілених обчислень Леслі Лампорта [1]: *«**Розподіленою системою** можна назвати таку систему, у якій відмова комп'ютера, про існування якого ви навіть не підозрювали, може зробити ваш власний комп'ютер непридатним до використання».*

Це означення він дав у травні 1987 року в своєму листі колегам з приводу чергового відключення електроенергії в машинному залі.

Трохи більш суворе означення, якого в подальшому будемо дотримуватися, запропонував Ендрю Таненбаум в своїй фундаментальній праці «Розподілені системи. Принципи та парадигми» [2]: ***«Розподілена система** – це набір з'єднаних каналами зв'язку незалежних комп'ютерів, які, з точки зору користувача деякого програмного забезпечення, мають вигляд єдиного цілого».*

У цьому формулюванні фіксуються два істотні моменти, характерні для РС:

- автономність вузлів;
- сприйняття системи користувачем як єдиної структури; при цьому основною сполучною ланкою розподілених систем є програмне забезпечення.

Розподілені системи створювалися і створюються для вирішення конкретного виду завдань, і вимоги до функціональності таких систем можуть значно різнитися, тому кількість розроблених досьогодні РС є дуже великою. Для спрощення розгляду РС можна поділити на кілька типів. Сьогодні використовують дві основні класифікації розподілених систем [3]:

1) за розмірами систем і способами адміністрування;
2) за функціональністю, тобто за типом наданих ресурсів і видами прикладних задач, під розв’язання яких їх оптимізовано.

За розмірами і способами адміністрування розрізняють такі РС:

- **кластери**, що містять до декількох десятків комп'ютерів, об'єднаних з допомогою локальної мережі, де адміністрування здійснюється вручну;
- **РС корпоративного рівня**, які містять десятки і навіть сотні комп'ютерів, для роботи яких необхідно встановлювати правила спільного використання ресурсів; через відносно невеликий масштаб у таких системах можна обходитися прямими адміністративними заходами для організації роботи ресурсів і користувачів;

- **глобальні системи**, що містять до декількох мільйонів комп'ютерів, розподілених по світу й об'єднаних глобальною мережею; в таких системах адміністративне програмне забезпечення (ПЗ) вбудовується в проміжне програмне забезпечення.

З огляду на функціональність серед РС можна виділити [4]:

- **обчислювальні системи**, у яких основним комп'ютерним ресурсом є потужність процесора;

- **інформаційні системи**, у яких основним комп'ютерним ресурсом є місткість пам'яті даних; такі системи можуть розглядатися як величезні сховища даних.

Слід зазначити, що нині практично неможливо визначити чіткі межі між цими типами систем, тому в подальшому буде використовуватися термін «розподілені обчислювальні системи» (РОС).

1.1. Проміжне програмне забезпечення

РОС є програмно-апаратним комплексом, орієнтованим на вирішення певних завдань. З одного боку, кожен обчислювальний вузол є автономним елементом, а з іншого – програмна складова РОС має забезпечувати користувачам видимість роботи з єдиною обчислювальною системою. У зв'язку з цим виділяють такі **важливі характеристики РОС** [5]:

1. Можливість роботи з пристроями різних типів:
   - наданими різними постачальниками;
   - що працюють на базі різних операційних систем;
   - що базуються на різних апаратних платформах.

Обчислювальні середовища, що складаються з безлічі обчислювальних систем на базі різних програмно-апаратних платформ, називаються ***гетерогенними***.

2. Можливість простого розширення і масштабування.

3. Постійна доступність ресурсів (навіть у випадку перебування деяких елементів РОС поза доступом).

4. Приховування особливостей комунікації від користувачів.

Для забезпечення роботи гетерогенного обладнання РОС у вигляді єдиного цілого стек ПЗ зазвичай розбивають на два шари (рис. 1.1). На верхньому шарі розташовуються розподілені додатки, що відповідають за розв’язання певних прикладних задач засобами РОС. Їх функціональні можливості базуються на нижньому шарі – проміжному програмному забезпеченні (ППЗ), яке взаємодіє з системним ПЗ і мережним рівнем для забезпечення прозорості роботи додатків у РОС.

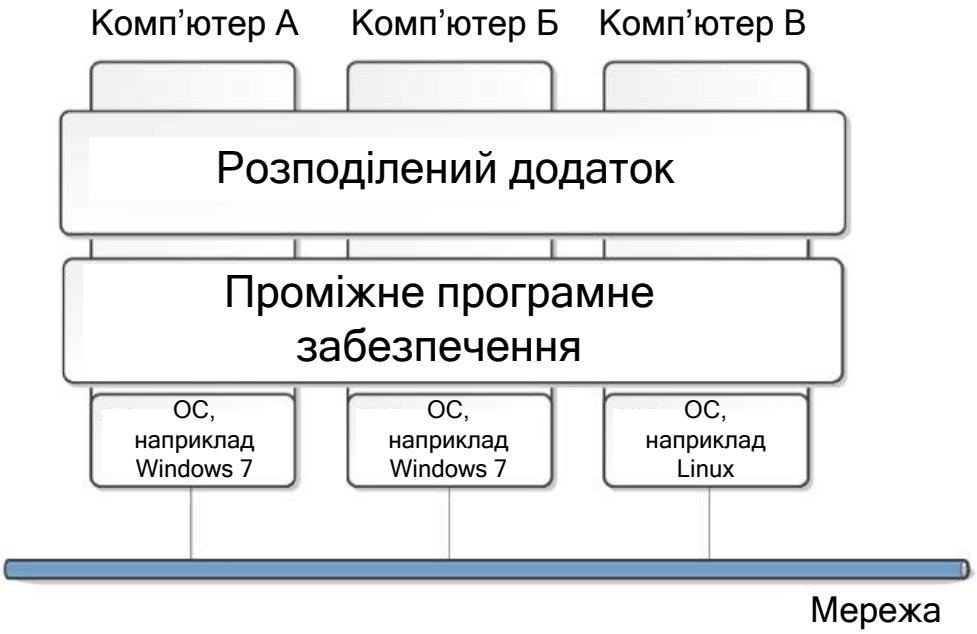

Рис. 1.1. Шари програмного забезпечення в РОС

Для того щоб РОС могла бути подана користувачеві як єдина система, застосовують такі типи прозорості в РОС:

- ***прозорий доступ до ресурсів*** – від користувачів має бути приховано різницю в поданні даних і в способах доступу до ресурсів РОС;
- ***прозоре місце розташування ресурсів*** – місце фізичного розташування необхідного ресурсу має бути несуттєвим для користувача;
- ***реплікація*** – приховування від користувача того, що в реальності існує більше однієї копії використовуваних ресурсів;
- ***паралельний доступ*** – можливість спільного (одночасного) використання одного й того ж ресурсу різними користувачами незалежно один від одного; при цьому факт спільного використання ресурсу має залишатися прихованим від користувача;
- ***прозорість відмов*** – відмова (відключення) будь-яких ресурсів РОС не має впливати на роботу користувача та його додатка.

1.2. Термінологія РОС

Ресурс – будь-яка програмна або апаратна сутність, присутня або використовувана у розподіленій мережі (наприклад, комп'ютер, пристрій зберігання, файл, комунікаційний канал, сервіс тощо).

Вузол – будь-який апаратний пристрій у розподіленій обчислювальній системі.

Сервер – це постачальник інформації у РОС (наприклад, веб-сервер).

Клієнт – це споживач інформації у РОС (наприклад, веб-браузер).

Пір – це вузол, який поєднує в собі як клієнтську, так і серверну частини (тобто і постачальника, і споживача інформації одночасно).

Сервіс – це мережна сутність, що надає певні функціональні можливості (наприклад, веб-сервер може надавати сервіс передачі файлів за протоколом HTTP). У межах одного вузла можуть надаватися кілька різних сервісів.

На рис. 1.2 показано схему взаємозв'язків між термінами РОС. Як видно, кожен комп'ютер або пристрій являє собою сутність у розподіленій обчислювальній системі у вигляді вузла. При цьому на кожному вузлі можуть розташовуватися кілька клієнтів, серверів, сервісів або пірів. Важливо зазначити, що будь-який вузол, сервер, пір або сервіс (але не клієнт!) є ресурсом РОС.

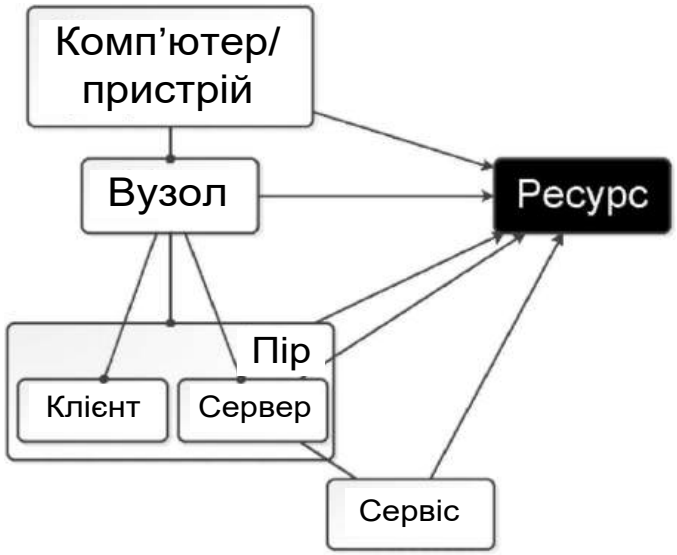

Рис. 1.2. Схема взаємозв'язків між термінами РОС

Сервіс отримує запит на надання певних даних (майже як аргументи, що передаються при виклику локальної функції) і повертає відповідь. Таким чином, сервіс можна визначити як певну заміну виклику функції на локальному комп'ютері. Існує безліч технологій, що забезпечують створення й супроводження сервісів у РОС: технологія XML веб-сервісів, сервіси REST тощо.

1.3. Класифікація РОС

РОС розрізняють за кількома класифікаційними ознаками: методом виявлення ресурсів, доступністю ресурсів, методом взаємодії ресурсів тощо. У межах кожної групи РОС класифікують за шкалою «централізований – децентралізований» [5]. Таким чином, залежно від використовуваного **методу виявлення** ресурсів виділяють РОС:

- *централізовані,* у яких запит на пошук відправляється на деякий центральний сервер; прикладом такої системи є служба DNS (Domain Name System – система доменних імен), яка за іменем сайту повертає його IP-адресу;
- *децентралізовані,* у яких запит на пошук відправляється всім вузлам, які відомі відправнику (наприклад, система Gnutella).

За доступністю ресурсів виділяють РОС:

- *централізовані, у* яких існує один сервер, що надає певний ресурс або сервіс (наприклад, веб-сервіс);
- *децентралізовані*, у яких кожен вузол відіграє роль клієнта і сервера й може надавати ресурси і сервіси, аналогічні іншим пристроям певної мережі (наприклад, BitTorrent, Gnutella, Napster).

За методом взаємодії ресурсів виділяють РОС:

- *централізовані, у* яких взаємодія між вузлами відбувається через центральний сервер, тобто один вузол не може звернутися до іншого безпосередньо;
- *децентралізовані,* у яких здійснюється пряма взаємодія між вузлами, оскільки кожен вузол відіграє роль сервера і клієнта (наприклад, однорангові обчислювальні системи).

1.4. Зв'язок у РОС

Поняття «розподілена обчислювальна система» означає, що компоненти такої системи розподілені, тобто віддалені один від одного. Очевидно, що функціонування подібних систем є неможливим без ефективного зв'язку між їх компонентами, який реалізується шляхом об'єднання пристроїв, що належать до РОС, у мережі.

Взаємодія в обчислювальних мережах базується на протоколах, що являють собою набори правил та угод, які описують процедуру взаємодії між компонентами системи.

В області обчислювальних комунікацій уже протягом тривалого часу існує загальноприйнята система протоколів – **мережна модель OSI** (англ. *Open Systems Interconnection basic reference model* – базова еталонна модель взаємодії відкритих систем), яка являє собою стек протоколів різного рівня (рис. 1.3).

Рис. 1.3. Рівні моделі OSI

1.5. Історія розвитку розподілених обчислень

Протягом усієї історії існування РОС ключовою проблемою їх розвитку був вибір між централізованою й розподіленою моделями надання обчислювальних ресурсів [5, 6].

Як тільки люди навчилися поєднувати комп'ютери між собою, багато груп дослідників почали вивчати можливості, що надаються системами розподілених обчислень, створюючи велику кількість бібліотек і проміжного програмного забезпечення, спрямованого на здійснення керування географічно розподіленими ресурсами таким чином, щоб вони були єдиною віртуальною паралельною обчислювальною машиною.

До середини 70-х років минулого століття через високу вартість телекомунікаційного обладнання й відносно слабку потужність обчислювальних систем домінувала централізована модель. Наприкінці 70-х років з появою систем поділу часу й віддалених терміналів намітилися передумови виникнення *клієнт-серверної архітектури*, що забезпечує надання ресурсів мейнфреймів (великих обчислювальних машин) кінцевим користувачам з допомогою віддаленого з'єднання. Подальший розвиток телекомунікаційних систем і поява персональних комп'ютерів дали поштовх розвитку клієнт-серверної парадигми оброблення даних.

На наступних етапах відбулися уточнення й переосмислення завдання розподілених обчислень. 2000 року Ян Фостер визначив завдання розподілених обчислювальних систем як «гнучкий, безпечний, координований розподіл ресурсів серед динамічних наборів користувачів, організацій і ресурсів» [7]. Він запропонував називати такі розподілені обчислювальні системи терміном *«грід»*. Комерційні розробники, розвиваючи ідею грід, у 2007–2008 роках представили концепцію *хмарних обчислень*, в основі якої було надання високомасштабованих віртуальних обчислювальних ресурсів кінцевому користувачеві через інтернет у вигляді послуг.

Нині використання розподілених обчислень у вигляді технологій грід і хмарних обчислень стає все більш популярним. Вони застосовуються для вирішення різних завдань, їх використання стає все простішим. Розподілені обчислення стають невід'ємною частиною наукових і комерційних високопродуктивних обчислень.

1.5.1. Перше покоління РОС

Перші проекти з розподілених обчислень, що виникли на початку 90-х років, ґрунтувалися на об'єднанні обчислювальних можливостей суперкомп'ютерів. Основною метою цих проектів було **надання обчислювальних ресурсів для певних високопродуктивних додатків**. Як типові про-

екти того часу можна розглядати проекти FAFNER [8] і I-WAY [9], які стали базовими для всієї галузі розподілених обчислень в подальшому. На них базувалися перші спроби стандартизації розподілених обчислень у гетерогенних обчислювальних середовищах.

Проект FAFNER (Factoring via Network-Enabled Recursion – мережне розкладання на множники з допомогою рекурсії) було створено 1995 року для вирішення завдання розкладання великих чисел на основі потужностей географічно розподілених обчислювальних систем (ОС). Знаходження простих множників великих чисел було необхідним для розшифровки даних, зашифрованих на основі алгоритму RSA [10]. Серед особливостей цього проекту слід зазначити таке:

- реалізація NFS (Network File System – мережна файлова система), яка давала змогу навіть малим робочим станціям (з 4 Мб оперативної пам'яті) виконувати корисну роботу, розраховуючи свій маленький фрагмент задачі;

- анонімна реєстрація учасників;

- ієрархічна структура веб-серверів, які являли собою кістяк ОС, що зменшувало можливість виникнення в ній «вузького місця».

I-WAY (Information Wide Area Year – рік інформації глобальних мереж) – експериментальна високопродуктивна мережа, спроектована на початку 1995 р. з метою об'єднання високошвидкісних мереж, що існували на той момент. У межах проекту було побудовано апаратну інфраструктуру, за допомогою якої здійснювався доступ до ресурсів мережі I-WAY. Вона складалася з базових робочих станцій під керуванням операційної системи UNIX, на які було встановлено спеціальне ППЗ (сервер I-POP). Система I-POP виконувала функції шлюзу до ресурсів мережі I-WAY. Кожен такий сервер підтримував стандартні процедури аутентифікації, резервування ресурсів, створення процесів і комунікації. Проект I-WAY використовувався для вирішення таких завдань [11]: суперкомп'ютерні обчислення; доступ до віддалених ресурсів; розв’язання задач віртуальної реальності.

1.5.2. Друге покоління РОС

Проекти високопродуктивних обчислювальних систем, реалізовані на початку 90-х років, виявили основні проблеми, які необхідно було вирішити для розгортання стабільних РОС. Для вирішення всіх зазначених завдань було розроблено концепцію ***грід*** – РОС, що забезпечує *«гнучкий, безпечний, координований розподіл ресурсів серед динамічних наборів користувачів, організацій і ресурсів»* [7]. Основним завданням грід була побудова

інфраструктури для здійснення *«обчислень на вимогу»*. Для того щоб називатися «грід», система має відповідати таким вимогам:

1. ***Гетерогенність.*** Обчислювальне середовище грід може складатися з безлічі різних ресурсів, що мають різні характеристики й параметри.

2. ***Масштабованість.*** Грід може складатися з як завгодно великої кількості ресурсів.

3. ***Пристосовність.*** Середовище грід може складатися з сотень комп'ютерів, і помилки в роботі десятка з них не мають впливати на отримані результати рішення.

Як приклад реалізації грід-концепцій можна назвати проект Globus [12], запущений 1997 року.

Наступним кроком у розвитку РОС стала поява систем, що базуються на ***об'єктно-орієнтованому підході***. Однією з перших таких систем став ***проект Legion*** [13], запущений у листопаді 1997 року, який надав програмну оболонку для організації однорідної взаємодії високопродуктивних гетерогенних РОС. Метою проекту було *надання користувачам єдиної інтегрованої інфраструктури РОС незалежно від масштабу, географічного положення, мови або операційної системи***.** Legion забезпечував користувача набором об'єктів, що надають базові сервіси. Основними серед них були:

- *об'єкти обчислювачів* – абстракції, що реалізують базові принципи роботи з обчислювальними ресурсами;

- *об'єкти систем зберігання даних* – абстракції, що надають базові методи роботи з системами зберігання даних;

- *об'єкти зв'язування* – об'єкти, що забезпечують зв'язок між абстрактним ідентифікатором об'єкта та його фізичною адресою;

- *об'єкти контексту* – об'єкти, що реалізують проекцію користувацьких імен об'єктів на абстрактні ідентифікатори об'єктів у системі Legion.

При розгляді другого покоління розподілених обчислювальних систем не можна залишити поза увагою такий клас, як ***розподілені об'єктні системи***. Вони надають базові методи для реєстрації, *серіалізації* (процес перетворення будь-якої структури даних на послідовність бітів) і *десеріалізації* (відновлення початкового стану структури даних з бітової послідовності) об'єктів, забезпечуючи віддалений виклик методів.

Серед представників цього класу слід виділити архітектуру CORBA [14] і модель Java [15].

CORBA (Common Object Request Broker Architecture – загальна архітектура брокера об'єктних запитів) – протокол взаємодії об'єктно-орієнтованих систем, опублікований у 1997–1998 роках консорціумом OMG (Object Management Group). Специфікація CORBA передбачає об'єднання програмного коду в об'єкт, який має містити інформацію про функціональ-

ність коду й інтерфейси доступу. Готові об'єкти можуть викликатися з інших програм (або об'єктів CORBA), розташованих у мережі.

Програмна платформа Java – сукупність програмних продуктів і специфікацій компанії Sun Microsystems (перший офіційний випуск – 23 травня 1995 року), для розроблення прикладного ПЗ і вбудовування його в будь-яке крос-платформне ПЗ. На відміну від CORBA, що використовує високорівневі стандарти взаємодії, в Java боротьба з гетерогенністю здійснюється шляхом використання єдиного віртуального середовища.

1.5.3. Сучасні РОС

Сьогодні РОС відхиляються від традиційних понять *високопродуктивних розподілених обчислень* у бік розвитку *віртуального співробітництва* й *віртуальних організацій*. *Віртуальна організація* – це група людей і/або організацій, об'єднаних спільними правилами колективного доступу до певних обчислювальних ресурсів [7]. Методи надання доступу до обчислювальних ресурсів стають сервісно-орієнтованими, тобто одні й ті самі ресурси можуть гнучко використовуватися різними споживачами. Крім того, значно розширилися області автоматизованого керування ресурсами й набули поширення автоматизовані засоби оброблення помилок і відновлення обчислювального процесу.

До сучасних РОС можна віднести однорангові мережі, сервіс-орієнтовану архітектуру, агентні системи й хмарні обчислення [5].

Однорангові мережі (peer-to-peer, P2P) виникли в 1999–2000 рр. У таких мережах комп'ютери обмінюються ресурсами безпосередньо один з одним, без використання центрального сервера. Застосування такої технології *спрощує масштабованість* мережі й *підвищує її відмовостійкість*, оскільки збій будь-якого обчислювального вузла не приводить до зупинки функціонування мережі цілком. До недоліків P2P можна віднести:

1) *підвищені вимоги до продуктивності* кожного комп'ютера, включеного в таку мережу*;*
2) *низький ступінь захищеності машин,* що беруть участь у P2P-мережі, через відкритий доступ до ресурсів;
3) необхідність подолання можливої *гетерогенності апаратного й програмного забезпечення* її потенційних учасників;
4) *трудомісткість і ресурсомісткість пошуку доступних ресурсів* без використання централізованої точки керування; як приклади P2P можна назвати проекти Gnutella, BitTorrent, Napster, Skype [5].

На початку 2000-х років бізнес-співтовариство почало розробляти специфікації з метою вирішення проблем ранніх стандартів розподілених об'єктних технологій з допомогою ***веб-сервісів*** і ***сервіс-орієнтованої архітектури (COA, Service-Oriented Architecture – SOA)***. Стандарти веб-

сервісів було розроблено за ініціативою організацій, що надають віддалений доступ до певних обчислювальних ресурсів. Їх закріплено консорціумом W3C. До основних стандартів розроблення та функціонування вебсервісів належать протоколи SOAP, WSDL і UDDI. Сьогодні сервіс-орієнтований підхід є стандартом де-факто при розробленні РОС.

З метою побудови широкомасштабних обчислювальних мереж, пристосованих до функціонування в навколишньому середовищі, що динамічно змінюється, з середини 90-х років почали розробляти ***агентні системи***. ***Програмний агент*** – це автономний процес, здатний реагувати на середовище виконання й спричиняти в ньому зміни, можливо, у кооперації з користувачами або іншими агентами.

Основними принципами роботи агентних мереж є:

- *автономність* – агенти функціонують автономно без можливості стороннього втручання в їх внутрішній стан;

- *соціальна поведінка* – агенти взаємодіють один з одним за допомогою певної мови (Agent Communication Languages, ACLs);

- *активність* – агенти взаємодіють з навколишнім середовищем, отримуючи певні сигнали й відповідаючи на них;

- *проактивність* – агенти діють цілеспрямовано.

Одним із найбільш відомих прикладів є архітектура взаємодії FIPA (Foundation for Intelligent Physical Agents – базис інтелектуальних фізичних агентів), яка стандартизує методи взаємодії агентів та агентних систем.

У 2007–2008 рр. почав набирати значущості термін ***«хмарні обчислення»***. Завданням таких систем є забезпечення повсюдного та зручного мережного доступу на вимогу до загального пулу конфігурованих обчислювальних ресурсів, які можуть бути оперативно надані та звільнені з мінімальними експлуатаційними витратами і/або зверненнями до провайдера. Сьогодні вже можна казати про те, що хмарні обчислення міцно увійшли в повсякденне життя кожного користувача інтернету (хоча багато хто про це й не підозрює). Однак досі немає єдиної думки про те, що таке хмарні обчислення і яким чином вони співвідносяться з парадигмою грід-обчислень.

2. АРХІТЕКТУРИ ІНФОРМАЦІЙНИХ СИСТЕМ

Інформаційна система (ІС) – організаційно впорядкована сукупність документів (масивів документів) та інформаційних технологій, у тому числі з використанням засобів обчислювальної техніки і зв'язку, що реалізують інформаційні процеси. Інформаційні системи призначено для зберігання, оброблення, пошуку, поширення, передачі та надання інформації.

Архітектура інформаційної системи – концепція, яка визначає модель, структуру, виконувані функції та взаємозв'язок компонентів інформаційної системи.

Якщо казати простими словами, інформаційна система – це сукупність ПЗ, яке розв’язує певну прикладну задачу.

Архітектура інформаційної системи – абстрактне поняття, що визначає, з яких складових частин (елементів, компонент) складається додаток і як ці частини між собою взаємодіють.

Під складовими частинами (елементами, компонентами) додатка зазвичай розуміють програми або програмні модулі, що виконують окремі, відносно ізольовані завдання.

Компоненти інформаційної системи за виконуваними функціями можна поділити на три шари (рис. 2.1):

- шар подання (інтерфейс користувача), що відповідає за взаємодію з користувачем: натискання кнопок, рух миші, виведення результатів пошуку тощо;

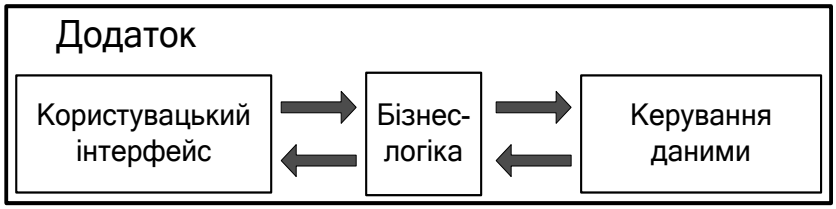

Рис. 2.1. Компоненти інформаційної системи

- бізнес-логіка, що містить правила, алгоритми реакції програми на дії користувача або на внутрішні події, правила оброблення даних;

- шар доступу до даних, що відповідає за зберігання, вибір, модифікацію й видалення даних, пов'язаних з прикладною задачею, що розв’язується певним додатком.

Традиційними архітектурами ІС є ***файл-серверна*** й ***клієнт-серверна***, які, своєю чергою, можуть бути дворівневими, 2,5-рівневими та трирівневими. При цьому еволюція архітектур ІС відбувалася в порядку «стоншення» клієнтів.

Під терміном ***«тонкий клієнт»*** мають на увазі багато пристроїв і програм, загальною властивістю яких є *можливість роботи в терміналь-ному режимі*. У загальному випадку тонкий клієнт (англ. *thin client*) являє собою комп'ютер або програму-клієнт у мережах з клієнт-серверною або термінальною архітектурою, які переносять усі або більшу частину завдань з оброблення інформації на сервер. Цим тонкий клієнт відрізняється від

товстого клієнта, який, навпаки, обробляє інформацію незалежно від сервера й використовує останній в основному для зберігання даних [17].

2.1. Файл-серверна архітектура

Файл-серверна архітектура (ФСА) виникла незабаром після появи локальних комп'ютерних мереж. У системах з такою архітектурою всі загальнодоступні файли зберігаються на виділеному комп'ютері – ***файл-сервері***.

Файл-серверні додатки – додатки, що використовують мережний ресурс для зберігання програми й даних. Модель файл-серверного додатка показано на рис. 2.2. Як видно, основною функцією сервера є зберігання даних і коду програми, у той час як функцією клієнта є оброблення даних.

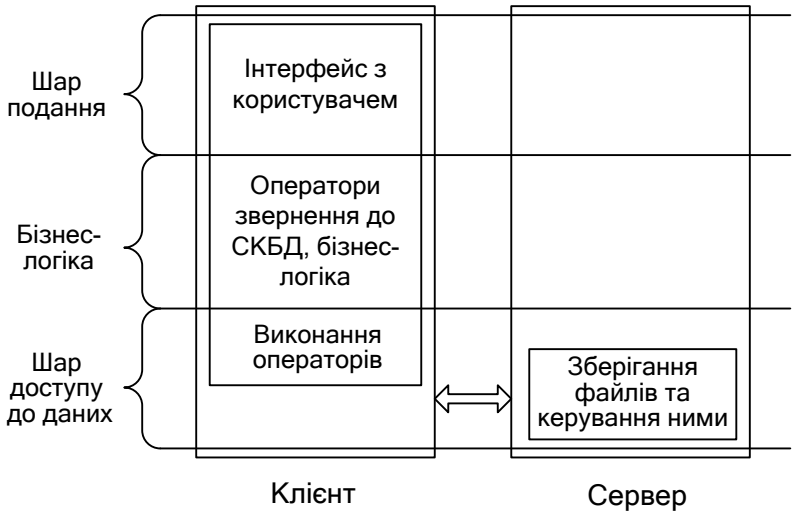

Рис. 2.2. Модель файл-серверного додатка

Очевидно, що клієнт у такій системі є товстим. При цьому кількість клієнтів обмежується десятками.

Переваги ФСА:

- можливість організації режиму роботи з даними, розрахованого на багато користувачів;
- зручність централізованого керування доступом;
- низька вартість розробки.

Недоліки ФСА: низькі продуктивність і надійність; слабкі можливості розширення.

Недоліки архітектури з файловим сервером пов'язані, головним чином, з тим, що дані зберігаються в одному місці, а обробляються в іншому. Їх потрібно передавати по мережі, що призводить до дуже високих навантажень на мережу й різкого зниження продуктивності додатка при збільшенні кількості клієнтів, що працюють одночасно. Іще одним важливим недоліком такої архітектури є децентралізоване вирішення проблем цілісності й узгодженості даних та одночасного доступу до них [18].

2.2. Клієнт-серверна архітектура

Історично першим варіантом клієнт-серверної архітектури (КСА) була так звана ***одноланкова*** архітектура, яка характеризується тим, що клієнт

не має ніякого функціонального навантаження, крім відображення інформації, що надається мейнфреймом (сервером). Прикладом такої архітектури може бути термінальний доступ до віддаленого сервера або віддалений робочий стіл. У цьому випадку весь обсяг обчислювального навантаження припадає на сервер.

Наступним кроком у розвитку КСА була поява так званої ***дволанкової (2-рівневої)*** архітектури (рис. 2.3). *Ключова відмінність КСА від ФСА* – абстрагування від фізичної схеми даних і маніпулювання даними клієнтськими програмами на рівні логічної схеми. Це дало змогу створювати надійні ІС, розраховані на багато користувачів, з централізованою базою даних (БД), незалежні від апаратної (а часто й програмної) частини сервера БД, які підтримують графічний інтерфейс користувача на клієнтських станціях, зв'язаних локальною мережею. ***Особливістю 2-рівневих КСА*** є те, що базові функції додатка розділені між клієнтом і сервером, при цьому клієнтська програма працює з даними через запити до серверного програмного забезпечення.

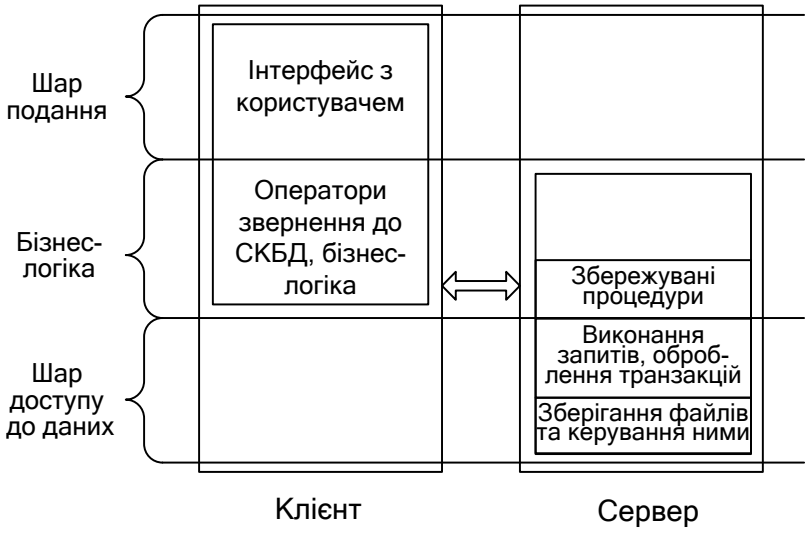

Рис. 2.3. Дволанкова КСА

Перевагами таких систем є:

- повна підтримка багатокористувацької роботи;
- гарантія цілісності даних.

Серед **недоліків** слід зазначити такі:

- необхідність оновлення ПЗ, призначеного для користувача, на кожному клієнті при будь-якій зміні алгоритмів, оскільки бізнес-логіка додатків залишилася в клієнтському ПЗ;
- високі вимоги до пропускної здатності комунікаційних каналів з сервером;
- слабкий захист даних від злому, особливо від недобросовісних користувачів системи;
- складність адміністрування й налаштування робочих місць користувачів системи;
- необхідність використання потужних ПК на клієнтських місцях;
- складність розроблення системи через необхідність виконувати бізнес-логіку й забезпечувати користувацький інтерфейс в одній програмі.

Неважко помітити, що більшість недоліків *класичної* або *2-рівневої* КСА пов'язані з тим, що клієнтська станція використовується як виконавець бізнес-логіки ІС. Тому очевидним кроком подальшої еволюції архітектур ІС стала ідея тонкого клієнта: алгоритми оброблення даних розбивалися на частини, пов'язані з виконанням бізнес-функцій і відображенням інформації в зручному для людини поданні; частина, пов'язана з первинною перевіркою й відображенням інформації, залишалася на клієнтській машині, а вся реальна функціональність системи переносилася на серверну частину.

На наступному етапі розвитку ІС виникла так звана **перехідна (2,5-рівнева)** КСА. Фізично така система складалася з двох компонентів: клієнта і сервера. Однак використання збережених процедур та обчислення даних здійснювалися на стороні сервера. При цьому обов'язковим стало використання систем керування базами даних (СКБД), що забезпечували багато переваг. Однак через написання програм для серверної частини на спеціалізованих вбудованих мовах СКБД, які не давали змоги написати всю бізнес-логіку додатка, частина бізнес-логіки все одно реалізовувалася на стороні клієнта, що призводило до його «потовщення».

Очевидними **перевагами 2,5-рівневої КСА порівняно з класичною** є:

- зниження вимог до швидкості передання даних між клієнтською і серверною частинами, оскільки обчислення реалізуються на серверній стороні, а мережею передаються вже готові результати обчислень;

- істотне поліпшення захисту інформації завдяки тому, що користувачам даються права на доступ до функцій системи, а не до її даних.

Серед **недоліків** слід зазначити таке:

а) обмежена масштабованість;

б) залежність від програмної платформи;

в) обмежене використання мережних обчислювальних ресурсів;

г) написання програм для серверної частини системи на слабо призначених для цього вбудованих у СКБД мовах опису процедур, що обумовлює:

- низьку швидкодію системи;

- високу трудомісткість створення й модифікації ІС;

- високу вартість апаратних засобів, необхідних для функціонування ІС.

Для усунення перелічених недоліків було запропоновано ***трирівневу КСА*** (рис. 2.4), основною відмінністю якої від 2,5-рівневої архітектури є фізичне розділення програм, що відповідають за зберігання даних (СКБД) і їх оброблення (сервер додатків (СД), application server (AS)). Такий поділ програмних компонент дає змогу оптимізувати навантаження як на мережне, так і на обчислювальне обладнання комплексу.

Переваги трирівневої КСА:

- клієнт є тонким;

- можливість передання між клієнтською програмою й СД лише мінімально необхідної кількості даних – аргументів функцій, які викли-

ються, і значень, які поветаються, що є теоретичною межею ефективності використання ліній зв'язку;

- можливість запуску СД в одному або декількох екземплярах на одному або декількох комп'ютерах, що дає змогу використовувати обчислювальні потужності організації настільки ефективно й безпечно, наскільки цього забажає адміністратор ІС;
- низька вартість трафіка між СД і СКБД; трафік між СД і СКБД може бути великим, проте це завжди трафік локальних мереж, а їх пропускна здатність є досить великою і дешевою; у крайньому випадку, СД і СКБД можна запустити на одній машині, що автоматично зведе мережний трафік до нуля;
- зниження навантаження на сервер даних порівняно з 2,5-рівневою схемою, а отже, і підвищення швидкості роботи системи в цілому;
- зниження витрат на нарощування функціональності й оновлення ПЗ.

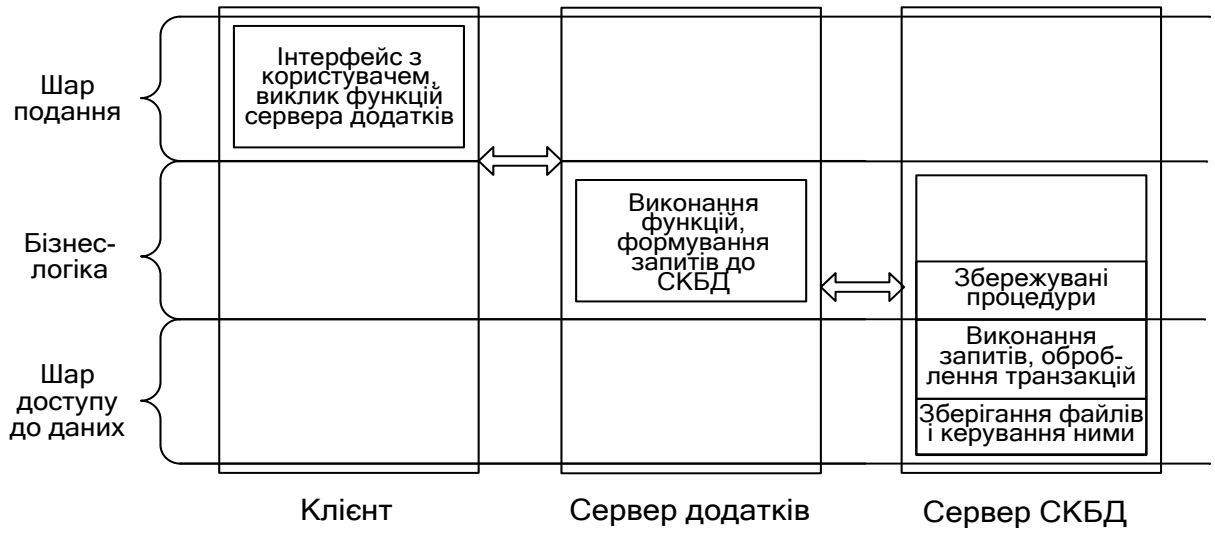

Рис. 2.4. Трирівнева КСА

Одним із недоліків цієї архітектури є підвищення витрат на адміністрування й обслуговування серверної частини.

Слід зазначити, що система з трирівневою КСА має практично необмежені можливості масштабування. Одна й та сама система може працювати як на одному окремому комп'ютері, виконуючи на ньому програми СКБД, СД і клієнтської частини, так і в мережі, що складається з сотень і тисяч машин. Єдиним фактором, що перешкоджає нескінченній масштабованості, є вимога ведення єдиної бази даних. Крім того, у ній істотно спростилося розширення функціональних можливостей, оскільки на відміну від 2,5-рівневої схеми при розширенні функціональності немає необхідності змінювати всю систему – достатньо встановити новий СД з

необхідною функцією. При цьому зменшується кількість проблем, пов'язаних зі встановленням заново клієнтських частин програми на безлічі комп'ютерів, які можуть бути дуже віддаленими, що було актуальним для систем з дворівневою КСА.

Як приклад системи з трирівневою КСА розглянемо пошукову машину в інтернеті, схему якої зображено на рис. 2.5. Користувач уводить рядок, який складається з ключових слів, й отримує список заголовків веб-сторінок. Результат формується з гігантської бази переглянутих і проіндексованих веб-сторінок. Ядром пошукової машини є програма, що трансформує введений користувачем рядок в один або кілька запитів до бази даних. Потім вона поміщає результати запиту в список і перетворює цей список на набір HTML-сторінок. Частина, яка відповідає за вибірку інформації, зазвичай знаходиться на рівні оброблення.

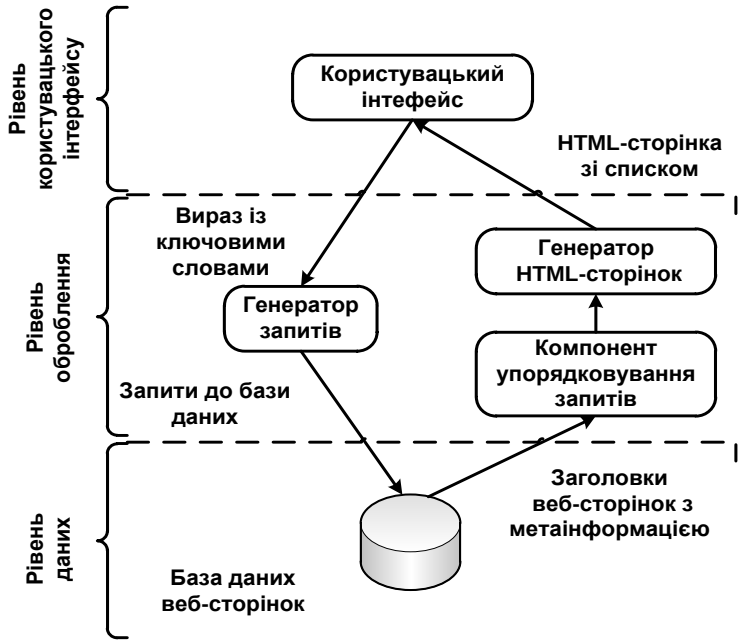

Рис. 2.5. Узагальнена організація пошукової машини для інтернету

Багатоланкові клієнт-серверні архітектури є прямим продовженням поділу додатків на рівні інтерфейсу користувача, компонентів оброблення й даних. Різні ланки взаємодіють відповідно до логічної організації додатка. У багатьох бізнес-додатках розподілене оброблення є еквівалентним організації багатоланкової архітектури додатків «клієнт – сервер». Такий тип розподілу називається ***вертикальним (ВР)***. Характерною особливістю вертикального розподілу є те, що він досягається *розміщенням логічно різних компонентів на різних машинах*.

У сучасних архітектурах розподіл на клієнти і сервери відбувається способом, відомим як ***горизонтальний розподіл (ГР)***. При такому типі розподілу клієнт або сервер може містити фізично розділені частини логічно однорідного модуля, причому робота з кожною з частин може відбуватися незалежно. Це робиться для вирівнювання навантаження.

Як приклад системи з ГР розглянемо веб-сервер, реплікований на кілька машин локальної мережі, схему якого зображено на рис. 2.6.

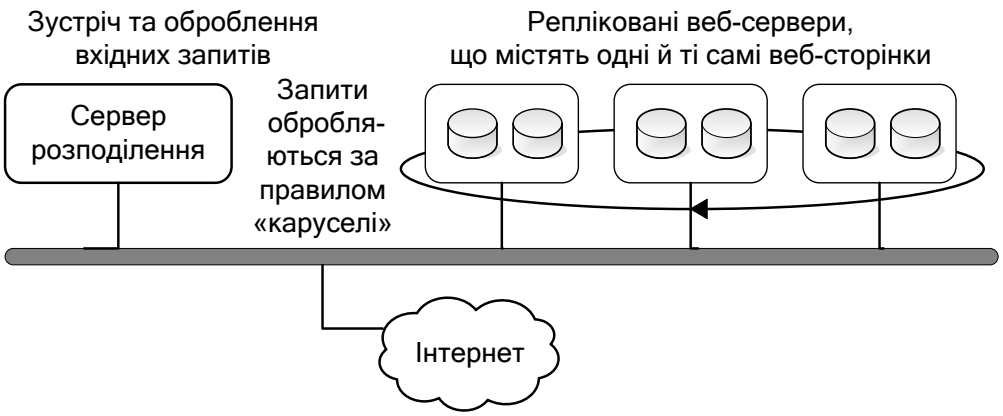

Рис. 2.6. Приклад ГР веб-сервера

Як видно, кожен із серверів містить один і той самий набір веб-сторінок. Коли одна з веб-сторінок оновлюється, її копії відразу розсилаються на всі сервери. Сервер, до якого буде передано запит, що приходить, вибирається за правилом «каруселі». Ця форма ГР успішно використовується для вирівнювання навантаження на сервери популярних веб-сайтів.

Таким же чином, хоча й менш очевидно, можуть бути розподілені й клієнти. Для нескладного додатка, призначеного для колективної роботи, сервер може взагалі бути відсутнім. У цьому випадку зазвичай кажуть про *одноранговий розподіл*. Найяскравішим прикладом застосування ГР є програмні системи, створені на основі концепції хмарних обчислень.

3. ОРГАНІЗАЦІЯ РОЗПОДІЛЕНИХ ОБЧИСЛЕНЬ З ВИКОРИСТАННЯМ СЕРВЕРІВ ДОДАТКІВ

Сервери додатків (СД) є однією з ключових складових IT-інфраструктури значної частини сучасних великих підприємств. Якщо компанія має потребу в інтеграції її внутрішньокорпоративних додатків з корпоративним Web-сайтом і Web-додатками, а також у надійному й швидкому доступі до даних і додатків з боку власних співробітників, партнерів і клієнтів, то вона рано чи пізно стикається з необхідністю впровадження одного або декількох СД.

Сервер додатків (англ. *application server*) – це програмна платформа, призначена для ефективного виконання процедур (програм, механічних операцій, скриптів), які підтримують побудову додатків. Сервер додатків діє як набір компонентів, доступних розробнику програмного забезпечення через API (інтерфейс прикладного програмування), який визначається самою платформою [18].

Переваги серверів додатків [19, 20]:

1. ***Цілісність даних і коду.*** Виділяючи бізнес-логіку на окремий сервер або на невелику кількість серверів, можна гарантувати оновлення й поліпшення додатків для всіх користувачів.

2. ***Централізоване налагодження й керування.*** Зміни в налагодженнях програми (наприклад, змінення сервера бази даних або установок системи) можуть проводитися централізовано.

3. ***Безпека.*** СД діє як центральна точка, використовуючи яку, постачальники сервісів можуть керувати доступом до даних і частин самих додатків. При цьому відповідальність за аутентифікацію можна перемістити з потенційно небезпечного рівня клієнта на рівень СД, додатково приховуючи рівень бази даних.

4. ***Підтримка транзакцій.*** *Транзакція* – одиниця активності, під час якої послідовне змінення ресурсів (в одному або різних джерелах) виконується атомарно (як неподільна одиниця роботи). Оскільки СД виконує велику кількість операцій з генерування коду, розробники можуть сфокусуватися на бізнес-логіці, що зменшує тривалість і вартість розроблення додатків.

5. ***Продуктивність.*** СД може вирішувати завдання балансування мережного трафіка й розподілу навантаження між іншими фізичними серверами системи.

Недоліками СД є:

1. ***Централізація*** – «падіння» СД або неполадки в мережному підключенні можуть призвести до недоступності програм для всіх клієнтів.

2. ***Захист інформації*** – проблема, пов'язана з передачею даних з використанням інфраструктури публічних мереж.

3.1. Архітектура сучасних корпоративних додатків

Сервер додатків – це інфраструктурне ПЗ, призначене для створення розподілених інформаційних систем з виділеними службами бізнес-логіки, реалізованими у вигляді компонентів, що виконуються під його керуванням. Ці компоненти можуть являти собою COM-об'єкти, CORBA-об'єкти або компоненти Enterprise JavaBeans.

До **функцій сервера додатків** належать:

- керування оптимізацією системних ресурсів (пам'яті, потоків та ін.);
- забезпечення зв'язку додатка з зовнішніми ресурсами (включаючи бази даних, мережі та інші додатки);
- забезпечення якісної підтримки сервісів (доступність, надійність, безпека, керованість, продуктивність, масштабованість);
- розгортання додатків (механізм їх поширення для встановлення на інших комп'ютерах, пристроях, серверах і в хмарі).

Сучасні СД дають змогу реалізувати надійні та стійкі до збоїв інформаційні системи шляхом підтримки створення кластерів та наявності засо-

бів відновлення після збоїв. СД є основою багатьох корпоративних рішень з підвищеними вимогами до надійності, наприклад, додатків, пов'язаних з електронною комерцією. Зазвичай при реалізації подібних рішень СД розташовуються між сервером баз даних (СБД) і Web-сервером або між СБД і клієнтськими додатками, при цьому іноді функціональність Web-сервера реалізується і в самому СД.

Сьогодні на корпоративному ринку домінують дві архітектури СД:

- *.NET* (тільки у виконанні Microsoft);

- *Java EE* (відома як J2EE), що надається безліччю компаній (мультивендорна).

Однак при цьому серйозну конкуренцію лідерам становлять нові програмні моделі, такі як Spring Framework, PHP, Ruby on Rails, Apex Code, Plain Old Java Object (POJO) [21].

Сервери додатків можуть бути доступними для користувачів:

- у вигляді продуктів, що встановлюються власне всередині компаній;
- у вигляді хмарних сервісів, що надаються провайдером.

3.1.1. Java Platform, Enterprise Edition

Java Platform, Enterprise Edition, скорочено **Java EE** (до версії 5.0 – Java 2 Enterprise Edition або J2EE) – набір специфікацій і відповідної документації для мови Java, що описує архітектуру серверної платформи для вирішення завдань середніх і великих підприємств.

Специфікації деталізовано настільки, що можна переносити програми з однієї реалізації платформи на іншу. Основна мета специфікацій – забезпечити *масштабованість* додатків і *цілісність даних* під час роботи системи. JEE багато в чому орієнтована на використання її через веб як в інтернеті, так і в локальних мережах. Уся специфікація створюється і затверджується через *JCP* (*Java Community Process*) у межах ініціативи Sun Microsystems Inc.

Популярності JEE також сприяє те, що Sun пропонує безкоштовний комплект розробки – SDK (Software Development Kit), що дає змогу підприємствам створювати свої системи без великих грошових витрат. До цього комплекту належить сервер додатків ***GlassFish*** з ліцензією для розроблення.

Версії наборів специфікацій Java EE (JEE) із зазначенням дат їх публікації наведено в табл. 3.1 [21]. Як видно, актуальна версія JEE має номер 9.0. При переході на версію 5.0 до набору специфікацій додалося кілька нових технологій і змінилася назва специфікації з J2EE (Java 2 Platform, Enterprise Edition) на Java Platform, Enterprise Edition, скорочено Java EE.

Java EE містить стандарти таких технологій:

- ***EJB*** – *Enterprise JavaBeans* – специфікація технології написання й підтримки серверних компонентів, що містять бізнес-логіку;

Таблиця 3.1

Версії наборів специфікацій JEE

Версія	Назва набору специфікацій	Дата публікації
1.0	Java 2 Platform Enterprise Edition, v 1.0	Грудень 1999 р.
1.2	Java 2 Platform Enterprise Edition, v 1.2	2000 р.
1.2.1	Java 2 Platform Enterprise Edition, v 1.2.1	23 травня 2000 р.
1.3	Java 2 Platform Enterprise Edition, v 1.3	24 вересня 2001 р.
1.4	Java 2 Platform Enterprise Edition, v 1.4	24 листопада 2003 р.
5.0	Java Platform, Enterprise Edition, v 5	11 травня 2006 р.
6.0	Java Platform, Enterprise Edition, v 6	6 грудня 2009 р.
7.0	Java Platform, Enterprise Edition, v 7	12 червня 2013 р.
8.0	Java Platform, Enterprise Edition, v 8	18 вересня 2017 р.
9.0	Java Platform, Enterprise Edition, v 9	2018 р.

- ***JPA*** – *Java Persistence API* – інтерфейс, що дає можливість зберігати в зручному вигляді Java-об'єкти в базі даних;

- ***Сервлет*** – клас, який розширює функціональні можливості сервера і взаємодіє з клієнтами за принципом «запит – відповідь»;

- ***JSP*** – *JavaServer Pages* – технологія, що дає змогу веб-розробникам легко створювати зміст, який має як статичні, так і динамічні компоненти. По суті сторінка JSP є текстовим документом, який містить текст двох типів: статичні вихідні дані, які можуть бути оформлені в одному з текстових форматів HTML, SVG, WML або XML, і JSP-елементи, які конструюють динамічний уміст;

- ***JSTL*** – *JavaServer Pages Standard Tag Library* – стандартна бібліотека тегів JSP, яка розширює специфікацію JSP, додаючи бібліотеку JSP тегів для загальних потреб, таких як розбір XML-даних, умовне оброблення, створення циклів і підтримка **інтернаціоналізації** (технологічні прийоми розроблення, що спрощують адаптацію продукту до мовних і культурних особливостей регіону, відмінного від того, у якому розроблявся продукт);

- ***JSF*** – *JavaServer Faces* – компонентний серверний фреймворк для розроблення веб-додатків на технології Java, призначений для полегшення розроблення інтерфейсів користувача (ІК) для JEE-додатків. Підхід JSF ґрунтується на використанні компонентів. Стан компонентів ІК зберігається, коли користувач запитує нову сторінку, і потім відновлюється, якщо запит повторюється;

- ***JAX-WS*** – *Java API for XML Web Services* – це прикладний програмний інтерфейс мови Java для створення веб-служб;

- ***JNDI*** – *Java Naming and Directory Interface* – це Java API, організований у вигляді служби каталогів, який дає змогу Java-клієнтам відкривати і переглядати дані й об'єкти за їхніми іменами;

- ***JMS*** – *Java Message Service* – стандарт проміжного програмного забезпечення для розсилання повідомлень, що дає змогу додаткам, виконаним на платформі Java EE, створювати, надсилати, отримувати й читати повідомлення;

- ***JTA*** – *Java Transaction API* – Java API для транзакцій. Визначає взаємодію між менеджером транзакцій та іншими учасниками розподіленої транзакційної системи;

- ***JAAS*** – *Java Authentication and Authorization Service* – реалізація в мові програмування Java стандарту системи інформаційної безпеки PAM. **Pluggable Authentication Modules** (PAM, Plug-in аутентифікації) – набір поділюваних бібліотек, які дають змогу інтегрувати різні низькорівневі методи аутентифікації у вигляді єдиного високорівневого API. Це дає можливість надати єдині механізми для керування, вбудовування прикладних програм у процес аутентифікації;

- ***JavaMail*** – це Java API, призначений для отримання й відправлення електронної пошти з використанням протоколів SMTP, POP3 та IMAP;

- ***JACC*** – *Java Authorization Contract for Containers* – це стандарт, який містить контракти безпеки між модулями СД і політики авторизації. Ці контракти визначають способи установлення, налагодження й використання провайдерів авторизації в рішеннях доступу;

- ***JCA*** – *J2EE Connector Architecture* – рішення на базі технології Java для з'єднання серверів додатків і корпоративних інформаційних систем у межах рішень інтеграції корпоративних додатків;

- ***JAF*** – *JavaBeans Activation Framework* – стандартне розширення для платформи Java, що дає змогу скористатися стандартними послугами: визначити тип довільного фрагмента даних; інкапсулювати доступ до нього; виявити доступні для нього операції; установити відповідний боб (bean) для виконання операції(й);

- ***StAX*** – *Streaming API for XML* – інтерфейс прикладного програмування для читання й запису XML-файлів;

- ***CDI*** – *Context and Dependency Injection* (упровадження контекстів і залежностей) – це набір послуг, які дають змогу розробникам використовувати JavaBeans з JSF у веб-додатках. CDI допомагає зв'язати рівень веб-вузлів і рівень транзакцій платформи JEE.

3.1.2. Microsoft .NET Framework

.NET Framework – програмна платформа, випущена компанією Microsoft 2002 року. Основою платформи є загальномовне середовище виконання ***Common Language Runtime (CLR)***, яке підходить для різних мов програмування. Функціональні можливості CLR є доступними в будь-яких мовах програмування, що використовують це середовище.

Уважається, що платформа .NET Framework стала відповіддю компанії Microsoft на створення платформи Java компанії Sun Microsystems (нині належить Oracle), яка на той час набула досить великої популярності [22].

Хоча .NET є патентованою технологією корпорації Microsoft та офіційно розрахована на роботу під операційними системами сімейства Microsoft Windows. Існують незалежні проекти (насамперед **Mono** і **Portable.NET)**, що дають змогу запускати програми .NET на деяких інших операційних системах.

Архітектуру .NET показано на рис. 3.1. Як видно, програма для .NET Framework, написана на будь-якій мові програмування, що підтримується, спочатку переводиться компілятором в єдиний для .NET проміжний байт-код **Common Intermediate Language (CIL)**. У термінах .NET описана операція має назву *«збирання»* (англ. *assembly*). Потім код або виконується віртуальною машиною **CLR**, або транслюється утилітою **NGen.exe** у виконуваний код для конкретного цільового процесора.

Використання віртуальної машини є переважним, оскільки позбавляє розробників необхідності піклуватися про особливості апаратної частини. У разі застосування віртуальної машини CLR, вбудований в неї JIT-компілятор «на льоту» (just in time) перетворює проміжний байт-код в ма-

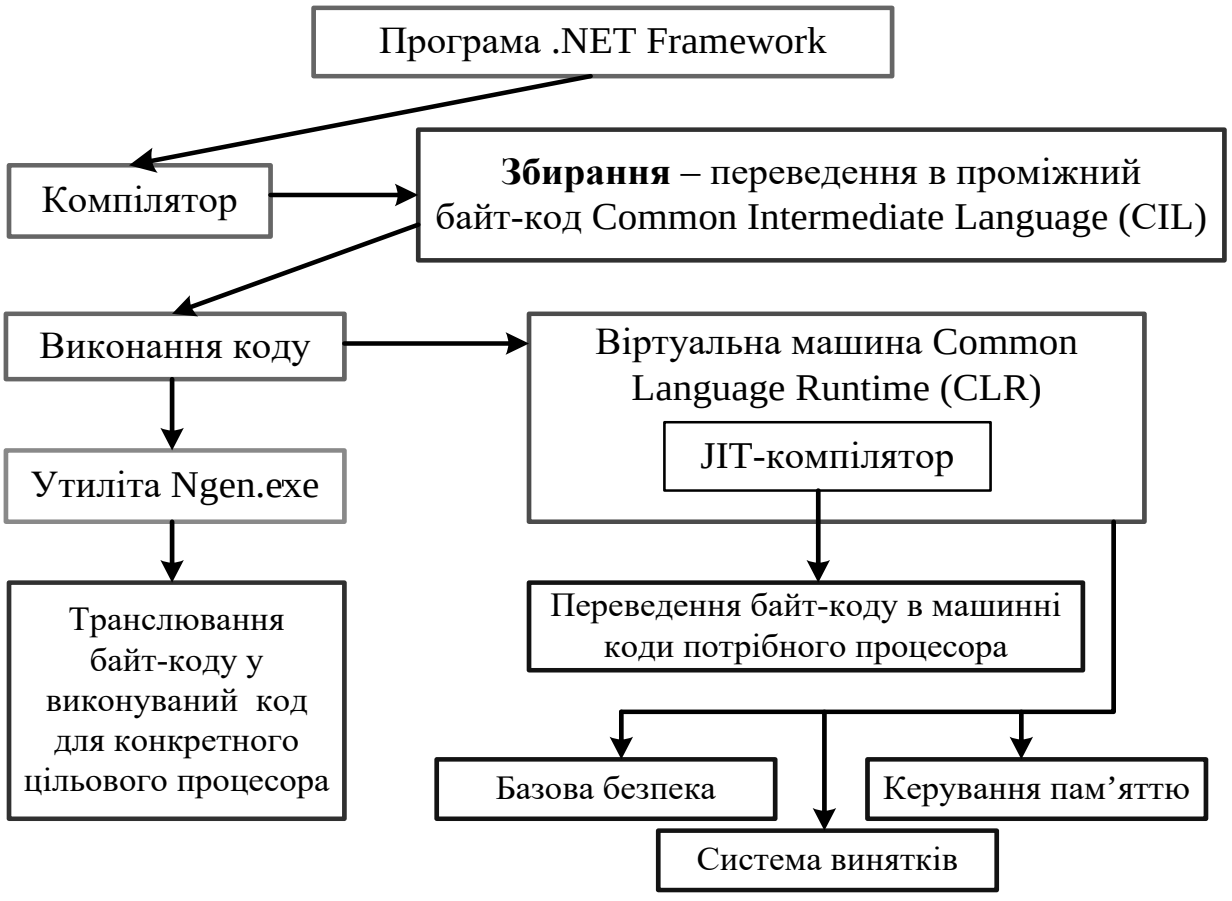

Рис. 3.1. Архітектура .NET

шинні коди потрібного процесора. При цьому використання сучасних технологій динамічної компіляції дає змогу досягти високого рівня швидкодії. Крім того, віртуальна машина CLR також сама піклується про базову безпеку, керування пам'яттю й систему винятків, позбавляючи розробника частини роботи.

Компанія Microsoft почала розробляти .NET Framework наприкінці 90-х під ім'ям Next Generation Windows Services (NGWS). 2000 року було випущено першу бета-версію .NET 1.0. Версії .NET із зазначенням дат їх виходу й операційними системами (ОС), у які відповідні версії .NET вбудовано за замовчуванням, наведено в табл. 3.2.

Таблиця 3.2

Версії платформи .NET Framework

Версія	Дата виходу	ОС, у яку версію вбудовано
1.0	1.05.2002	-
1.1	1.04.2003	Windows Server 2003
2.0	11.07.2005	Windows Vista, Windows 7, Windows Server 2008 R2
3.0	6.11.2006	Windows Vista, Windows Server 2008, Windows 7, Windows Server 2008 R2
3.5	9.11.2007	Windows 7, Windows Server 2008 R2
4.0	12.04.2010	Windows 8, Windows Server 2012
4.5	15.08.2012	Windows 8, Windows Server 2012
4.5.1	17.10.2013	Windows 8.1, Windows Server 2012 R2
4.5.2	5.05.2014	
4.6	20.07.2015	Windows 10
4.6.1	17.11.2015	Windows 10 v1511
4.6.2	20.06.2016	Windows 10 v1607
4.7	5.04.2017	Windows 10 v1703
4.7.1	17.10.2017	Windows 10 v1709, Windows Server 2016 (version 1709)
4.7.2	30.04.2018	Windows 10 v1803

Архітектуру .NET Framework описано й опубліковано в специфікації **Common Language Infrastructure (CLI)**, розробленій Microsoft і затвердженій *ISO* (Міжнародна організація зі стандартизації) і *Ecma* (раніше ECMA – European Computer Manufacturers Association – організація зі стандартизації інформаційних і комунікаційних технологій).

У CLI описано типи даних .NET, формат метаданих про структуру програми, систему виконання байт-коду та ін.

Стек протоколів, що входять у .NET, зображено на рис. 3.2. Об'єктні класи .NET, доступні для всіх підтримуваних мов програмування, містяться в бібліотеці **Framework Class Library (FCL)**. Ядро FCL називається **Base Class Library (BCL)**.

Windows Forms – API, який відповідає за графічний інтерфейс користувача.

ASP.NET (Active Server Pages) – технологія створення веб-додатків і веб-сервісів.

ADO.NET – технологія, що надає .NET-додаткам доступ до даних.

Windows Presentation Foundation (WPF) – система для побудови клієнтських додатків Windows з візуально привабливими можливостями взаємодії з користувачем. Програми, що розробляються, можуть бути автономними або запускатися в браузері.

Windows Communication Foundation (WCF) – програмний фреймворк, який використовується для обміну даними між додатками. До випуску в складі .NET Framework 3.0 був відомий під кодовим ім'ям **Indigo**. WCF дає змогу створювати безпечні й надійні транзакційні системи через спрощену уніфіковану програмну модель міжплатформної взаємодії. У WCF закладено принципи інтероперабельності, які дають змогу організовувати роботу з іншими платформами.

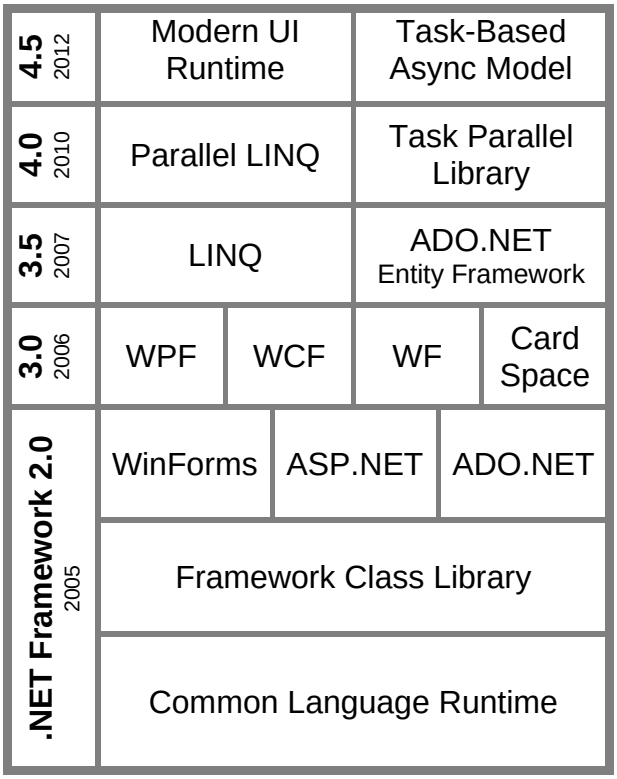

Рис. 3.2. Стек протоколів .NET

Windows Workflow Foundation (WF) – технологія для визначення, виконання робочих процесів (англ. *workflow*) і керування ними (**робочий процес (потік):** 1. Графічне зображення процесу виконання завдання й пов'язаних з ним підпроцесів. 2. Спосіб надходження інформації до різних об'єктів, які беруть участь у процесі). WF є орієнтованою на візуальне програмування й використовує декларативну модель програмування.

Windows CardSpace (WCS) – це спосіб ідентифікації користувачів при переміщенні між ресурсами інтернету без необхідності повторного введення імен і паролів. 15 лютого 2011 року корпорація Microsoft оголосила про скасування Windows CardSpace 2.0 і роботу над заміщувальним ПЗ U-Prove.

Language Integrated Query (LINQ) – проект з додавання синтаксису мови запитів, подібного до SQL (*Structured Query Language* – мова структурованих запитів), у мови програмування платформи .NET.

ADO.NET Entity Framework (EF) – об'єктно-орієнтована технологія доступу до даних, яка дає можливість взаємодії з об'єктами як з допомогою LINQ (LINQ to Entities), так і з використанням Entity SQL.

Паралельний LINQ (PLINQ) – паралельна реалізація LINQ to Objects, що надає повний набір стандартних операторів запиту LINQ у вигляді розширення для простору імен T: System.Linq і має додаткові оператори для паралельних операцій. PLINQ може значно збільшити швидкість за-

питів LINQ to Objects, ефективніше використовуючи всі доступні ядра на головному комп'ютері.

Task Parallel Library (TPL) - бібліотека паралельних задач, що являє собою поєднання загальних типів і API, які дають змогу реалізовувати паралелізм і узгодженість операцій. TPL є високорівневим каркасом паралельного програмування для .NET-коду, що дає змогу максимально використовувати продуктивність багатоядерних процесорів. TPL дає можливість реалізувати логічний паралелізм (указати те, що потенційно може працювати паралельно) замість жорсткого поділу на потоки. Реальне розпаралелювання здійснюється під час виконання завдання залежно від доступних апаратних засобів.

Modern UI Runtime – дизайнерський стиль, орієнтований на друкарське оформлення інтерфейсу користувача.

Task-based Asynchronous Pattern (TAP) – асинхронна модель, що базується на завданнях, – схема програмування, спрямована на створення асинхронних потоків у процесі виконання програми й керування ними.

Однією з основних ідей Microsoft .NET є сумісність програмних частин, написаних різними мовами. Наприклад, служба, написана мовою C ++ для Microsoft .NET, може звернутися до методу класу з бібліотеки, написаної мовою Delphi. Кожна бібліотека (збирання) в .NET має відомості про свою версію, що дозволяє усунути можливі конфлікти між різними версіями збирань.

Середовища розробки, що підтримують .NET: Microsoft Visual Studio (C#, Visual Basic .NET, Managed C++, F#), SharpDevelop, MonoDevelop, Embarcadero RAD Studio (Delphi for .NET) (раніше – Borland Developer Studio (Delphi for .NET, C#)), Zonnon, PascalABC.NET. Додатки .NET також можна розробляти в текстовому редакторі, просто викликаючи компілятор з командного рядка.

Головною проблемою .NET є орієнтованість тільки на підтримку ОС сімейства Microsoft і використання продуктів від одного постачальника. У зв'язку з цим було розпочато кілька проектів зі створення повноцінного втілення системи .NET Framework на базі вільного програмного забезпечення. Одним з таких проектів є **Mono**.

Основний розробник проекту Mono – компанія Xamarin (раніше – Novell). Після укладення договору Microsoft з Novell платформу Mono було офіційно визнано реалізацією .NET на Unix-подібних операційних системах: Linux, Mac OS X та ін. (хоча Mono успішно працює і під Microsoft Windows). Однак договір стосується тільки Novell і клієнтів Novell. Крім того, технології ASP.NET, ADO.NET і Windows Forms були стандартизовані ECMA / ISO, тому їх використання в Mono перебуває під загрозою юридичних претензій з боку Microsoft. У зв'язку з цим Mono надає реалізації ASP.NET, ADO.NET і Windows.Forms, проте рекомендує не використовувати ці API.

3.2. Лідери ринку серверів додатків

На думку дослідної та консалтингової компанії Gartner, що спеціалізується на ринках інформаційних технологій (IT), СД можуть застосовуватися **в трьох основних сценаріях** [23]:

1. *Тимчасово орієнтовані проекти* – швидке розроблення і розгортання додатків у відповідь на бізнес-вимоги, які "не можуть чекати". У цьому випадку час реалізації проекту є більш важливим, ніж забезпечення доступності, масштабованості тощо.

2. *Проекти масового ринку* – не дуже складні прикладні системи, які створюються невеликими IT-компаніями. Тут важливими є такі параметри, як низька вартість, простота розгортання, підтримки й керування, надійність. Функціональність, масштабованість і продуктивність є не настільки істотними.

3. *Систематично орієнтовані проекти* – реалізація критично важливих для бізнесу корпоративних систем, розрахованих на довгостроковий період експлуатації (не менше трьох років). Поряд з різноманітною функціональністю тут необхідними є надійність, безпека, керованість, масштабованість, продуктивність.

Згідно з даними Gartner, лідерами ринку СД масштабу підприємства (EAS – Enterprise Application Server) є *IBM, Microsoft, Oracle, Red Hat (JBoss)*, проте найближчим часом до них приєднаються компанії *Google* і *Tibco* зі своїми рішеннями *App Engine* і *Silver* відповідно [23].

3.2.1. IBM

Сімейство ПЗ **IBM WebSphere** містить великий набір EAS-пропозицій (у т. ч. серію продуктів **WebSphere Application Server – WAS**), який задовольняє широкий діапазон вимог замовників:

а) для тимчасово орієнтованих проектів (*WebSphere sMash*);

б) для масового ринку (*WAS Community Edition, WAS Express*);

в) для масштабованих корпоративних рішень:

- *WAS Network Deployment;*
- *WebSphere Virtual Enterprise;*
- *CloudBurst Appliance* та ін.

Деякі з цих засобів є доступними у вигляді **хмарних сервісів IBM** і **Amazon Web Services EC2**.

WAS-рішення базуються в цілому на стандартах Java EE і SOA, забезпечуючи при цьому підтримку різних моделей програмування. Слід зазначити, що IBM має в своєму розпорядженні потужні набори засобів розроблення (Rational) і керування IT (Tivoli).

Проте присутність IBM на масовому ринку поки є невеликою. Випущений в середині 2008 року WebSphere sMash поки має відносно невелику інстальовану базу й досить обмежену підтримку з боку третіх фірм. Про-

дукти для хмар (WAS Hypervisor Edition, CloudBurst Appliance) також з'явилися відносно недавно, і поки в цій сфері IBM помітно відстає від лідерів.

3.2.2. Microsoft

.NET Framework разом з **Internet Information Server** (обидва є інтегрованими компонентами **Windows Server**) являють собою повний набір функціональності EAS. Продукти сімейства корпоративних серверів **Microsoft Server System** призначені, як і інші СД, для створення і розгортання інтегрованих корпоративних рішень.

При відносно невисокій вартості для цих СД є характерними:

- підтримка XML;
- підтримка стандартів інтернету;
- підтримка кластерної архітектури;
- високий ступінь взаємної інтеграції.

За функціональністю сімейство серверів Microsoft Server System охоплює практично всі сучасні області застосування СД. Усі сервери сімейства Microsoft Server System підтримують керування COM-, COM+- і .NET-компонентами і є доступними для операційних систем сімейства Windows, проте для інших платформ ці продукти не випускалися і не випускаються.

З продуктів, що належать до сімейства Microsoft Server System, до серверів додатків у традиційному розумінні можна віднести:

- сервер інтеграції додатків *Microsoft BizTalk Server*;
- сервер повідомлень і групової роботи *Microsoft Exchange Server;*
- сервер електронної комерції *Microsoft Commerce Server;*
- масштабований сервер додатків для мобільної телефонії *Microsoft Mobile Information Server*;
- корпоративний портал *Microsoft SharePoint Portal Server*;
- сервер для керування інформаційним наповненням Web-сайтів *Microsoft Content Manager Server*;
- сервер для керування великими корпоративними проектами *Microsoft Project Server*.

Хмарні пропозиції Microsoft реалізовано у вигляді бета-версії **Windows Azure Platform** і недавно анонсованої технології програмованої хмарної платформи **xRM**.

Переваги Microsoft – величезна інстальована база Windows Server і велике співтовариство розробників ПЗ, що робить її продукти стандартом де-факто на масовому ринку. При цьому компанія постійно нарощує свою присутність у сфері великих корпоративних проектів і має дуже чітку стратегію реалізації хмарної моделі обчислень.

Недоліки Microsoft – підтримка однієї ОС і використання продуктів від одного постачальника. Через традиційне фокусування на масовому ринку компанія постійно запізнюється з реалізацією важливих техно-

логічних ініціатив (наприклад, XTP (Extreme Transaction Processing – форма оброблення транзакцій в інформаційних технологіях) і SOA). При цьому Microsoft має кілька дуже серйозних суперників (Google, Salesforce, VMware), які також орієнтовані на масовий ринок, але в конкурентній боротьбі використовують інші ділові й технологічні моделі.

3.2.3. Oracle

Основою EAS-пропозицій Oracle є JEE-сімейство **Oracle WebLogic Server (WLS)**, отримане внаслідок придбання 2008 року компанії BEA Systems, і власна розробка **Oracle Application Server** (вона ще підтримується, але в стратегічному плані не розвивається). Крім того, до групи EAS-продуктів належать: засіб розробки Oracle JDeveloper; інструмент Oracle TopLink; засіб для моніторингу, адміністрування й керування Oracle Enterprise Manager.

Унаслідок придбання Sun Microsystems компанія поповнила свій EAS-арсенал ще й цілим портфелем EAS-технологій (зокрема, вільним середовищем розробки NetBeans), завдяки чому стала провідною в Java-співтоваристві.

WLS-сімейство в цілому фокусується на підтримці потужних критично важливих бізнес-додатків, але при цьому відповідає вимогам широкого спектра клієнтів, у тому числі з малого і середнього бізнесу. Ці рішення широко застосовуються на ринку і ґрунтуються на найсучасніших технологічних концепціях.

Сьогодні в Oracle слабо представлені EAS-рішення, орієнтовані на масовий ринок, хоча придбання Sun GlassFish частково заповнило цю прогалину в спектрі ПЗ Oracle.

Компанію Oracle часто критикують за занадто активну стратегію придбань, яка часом ставить у глухий кут замовників Oracle, які перестають орієнтуватися в перспективах розвитку того або іншого напрямку продуктів компанії.

3.2.4. Red Hat

JBoss EAS – це JEE5-сумісний **JBoss Application Server** (з квітня 2013 р. – **WildFly**), доступний для безкоштовного завантаження (без технічної підтримки) або в складі пакета для підприємств у вигляді **JBoss Enterprise Application Platform** з попередньою оплатою підтримки.

Крім того, Red Hat має набір JBoss Enterprise SOA Platform, до якого входять сервер додатків і багато технологій підтримки SOA, включаючи рішення **JBoss ESB**, призначене для інтеграції різних систем, у тому числі несумісних.

Є також EAS-пропозиція **JBoss Communications Platform**, орієнтована на використання в телекомунікаційній галузі.

Для Web-проектів пропонується **JBoss Enterprise Web Server**, що містить сервер додатків Apache Tomcat.

До сімейства ПО JBoss належить багато інших продуктів та інструментів, у тому числі засоби розроблення й керування IT-інфраструктурою.

Усе це ПЗ, *безкоштовне* або таке, що *отримується за передплатою*, поширюється за ліцензіями LGPL 2.x або Apache Software і є доступним у вигляді вихідних кодів або об'єктних модулів.

JBoss EAS має високу технічну репутацію на ринку, а Red Hat є явним лідером серед постачальників відкритих EAS-рішень. Її продукти мають величезну інстальовану базу й велику кількість партнерів і користувачів. Фактично це єдине **Open Source**-сімейство на IT-ринку, яке на рівних конкурує з пропозиціями провідних пропрієтарних вендорів. Але іноді результатом бізнес-стратегії Red Hat, спрямованої на підвищення прибутковості підрозділу JBoss, є уповільнення впровадження інженерних інновацій. Компанія явно відстає від конкурентів в освоєнні передових технологій, таких як *XTP*, *подієве керування* і *хмарні обчислення*.

4. ВІДДАЛЕНИЙ ВИКЛИК ПРОЦЕДУР

Ідея виклику віддалених процедур (Remote Procedure Call – RPC) полягає в адаптації механізму передання керування й даних усередині програми, що виконується на одній машині, до ситуації, коли керування й дані мають передаватися по мережі [5].

Використання засобів віддаленого виклику процедур дає змогу:

- полегшити організацію розподілених обчислень;
- створювати розподілені клієнт-серверні інформаційні системи.

RPC найбільш ефективно використовується в тих додатках, у яких існує інтерактивний зв'язок між віддаленими компонентами з невеликим часом відповідей і відносно малою кількістю даних, що передаються. Такі додатки називають ***RPC-орієнтованими***.

Характерними рисами виклику віддалених процедур є:

- асиметричність – одна із взаємодійних сторін є ініціатором;
- синхронність – виконання процедури, що викликається, припиняється з моменту видачі запиту і відновлюється тільки після повернення з процедури, що визивалася.

Проблеми й завдання, які необхідно вирішити при реалізації RPC:

1. Процедура, що викликає, і процедура, що викликається, виконуються на різних машинах і мають різні адресні простори, що створює проблеми при переданні параметрів і результатів, особливо якщо машини перебувають під керуванням різних операційних систем або мають різну архітектуру.

2. На відміну від локального виклику віддалений виклик процедур обов'язково використовує транспортний рівень мережної архітектури (наприклад, TCP), проте це залишається прихованим від розробника.

3. У реалізації RPC беруть участь щонайменше два процеси – по одному в кожній машині. У разі, якщо один з них аварійно завершиться, можуть виникнути такі ситуації:

- при аварії процедури, що викликає, процедури, що віддалено викликаються, стануть *«осиротілими»*;

- при аварійному завершенні віддалених процедур процедури, що викликають, стануть *«знедоленими батьками»*, які будуть безрезультатно чекати відповіді від віддалених процедур.

4. Неоднорідність мов програмування й операційних середовищ призводить до появи проблеми сумісності, яку досі не було вирішено ні за допомогою введення одного загальноприйнятого стандарту, ні за допомогою реалізації кількох конкуруючих стандартів на всіх архітектурах і у всіх мовах.

4.1. Базові операції RPC

Щоб зрозуміти роботу RPC, розглянемо спочатку виконання виклику локальної процедури в звичайному комп'ютері, що працює автономно. Щоб здійснити виклик, процедура, що викликає, поміщає параметри в стек у зворотному порядку. Після того як виклик виконано, вона поміщає значення, що повертається, у регістр, переміщує адресу повернення і повертає керування процедурі, що викликається, яка вибирає параметри зі стеку, повертаючи його в початковий стан.

Ідея, яка є основою RPC, полягає в тому, щоб виклик віддаленої процедури, якщо можливо, мав такий самий вигляд, як і виклик локальної процедури. Іншими словами, процедурі, що викликає, не потрібно знати про те, що процедура, яка викликається, знаходиться на іншій машині, і навпаки. Схему взаємодії програмних компонентів при виконанні віддаленого виклику процедури зображено на рис. 4.1.

Досягнення прозорості RPC. Коли процедура, що викликається, дійсно є віддаленою, у бібліотеку замість локальної процедури поміщається інша версія процедури, яку називають клієнтським ***стабом*** (англ. *stub* – заглушка). Як і оригінальна процедура, стаб викликається з використанням послідовності, що викликає, так само відбувається переривання при зверненні до ядра. Але на відміну від оригінальної процедури стаб не поміщає параметри в регістри і не запитує у ядра дані, а формує повідомлення для відправлення ядру віддаленої машини.

Основні етапи виконання процедури RPC показано на рис. 4.2. Після того, як клієнтський стаб був викликаний програмою-клієнтом, його першим завданням є ***заповнення буфера повідомленням, що відправляється***. У деяких системах клієнтський стаб має єдиний буфер фіксованої довжи-

ни, що заповнюється кожного разу з самого початку при надходженні кожного нового запиту. В інших системах буфер повідомлення являє собою пул буферів для окремих полів повідомлення, причому деякі з цих буферів є вже заповненими.

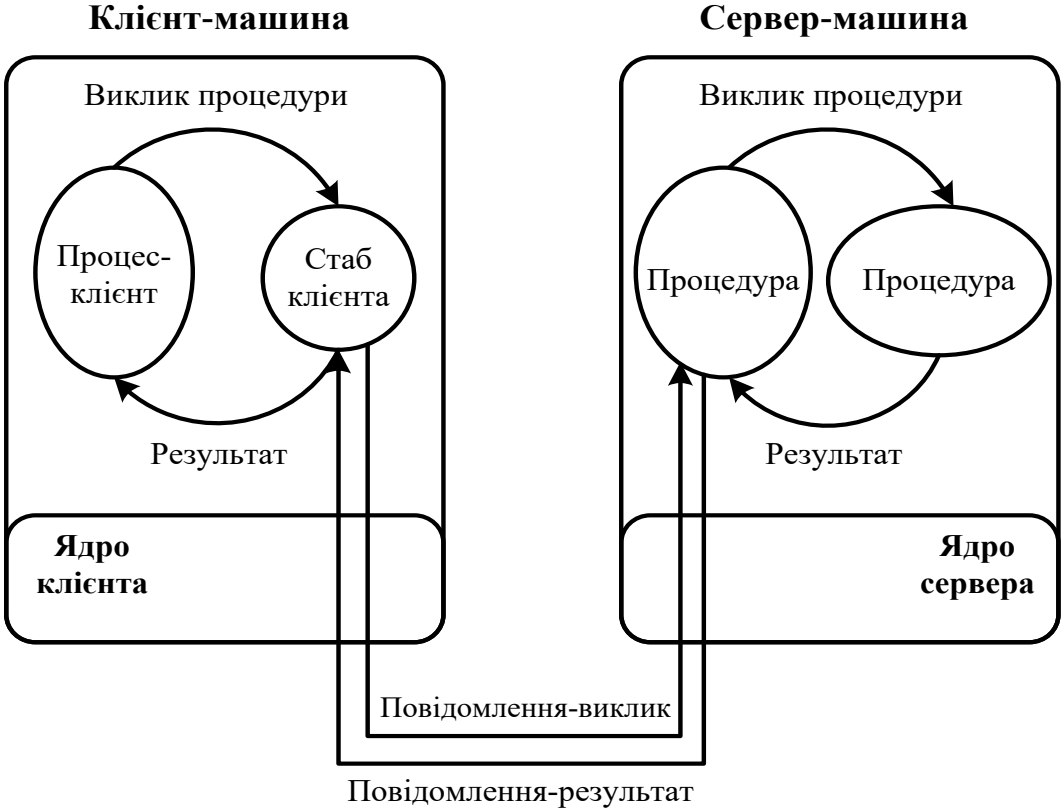

Рис. 4.1. Взаємодія програмних компонентів при виконанні віддаленого виклику процедури

Потім виконуються ***перетворення параметрів у відповідний формат*** та їх ***вставлення в буфер повідомлення***. До цього моменту повідомлення готове до передання, тому виконується ***переривання за викликом ядра***.

Коли ядро отримує керування, воно перемикає контексти, зберігає регістри процесора й карту пам'яті (дескриптори сторінок), установлює нову карту пам'яті, яка буде використовуватися для роботи в режимі ядра. Оскільки контексти ядра й користувача різняться, ядро має скопіювати повідомлення в свій власний адресний простір, запам'ятати ***адресу призначення***, після чого передати її ***мережному інтерфейсу***.

На цьому завершується робота на клієнтській стороні. Включається ***таймер передання***, і ядро може або почати виконувати циклічне опитування наявності відповіді, або передати керування планувальникові, який вибере будь-який інший процес на виконання.

На стороні сервера ***біти, що надходять, поміщаються*** приймальною апаратурою або ***у вбудований буфер, або в оперативну пам'ять***. Коли всю інформацію отримано, генерується переривання.

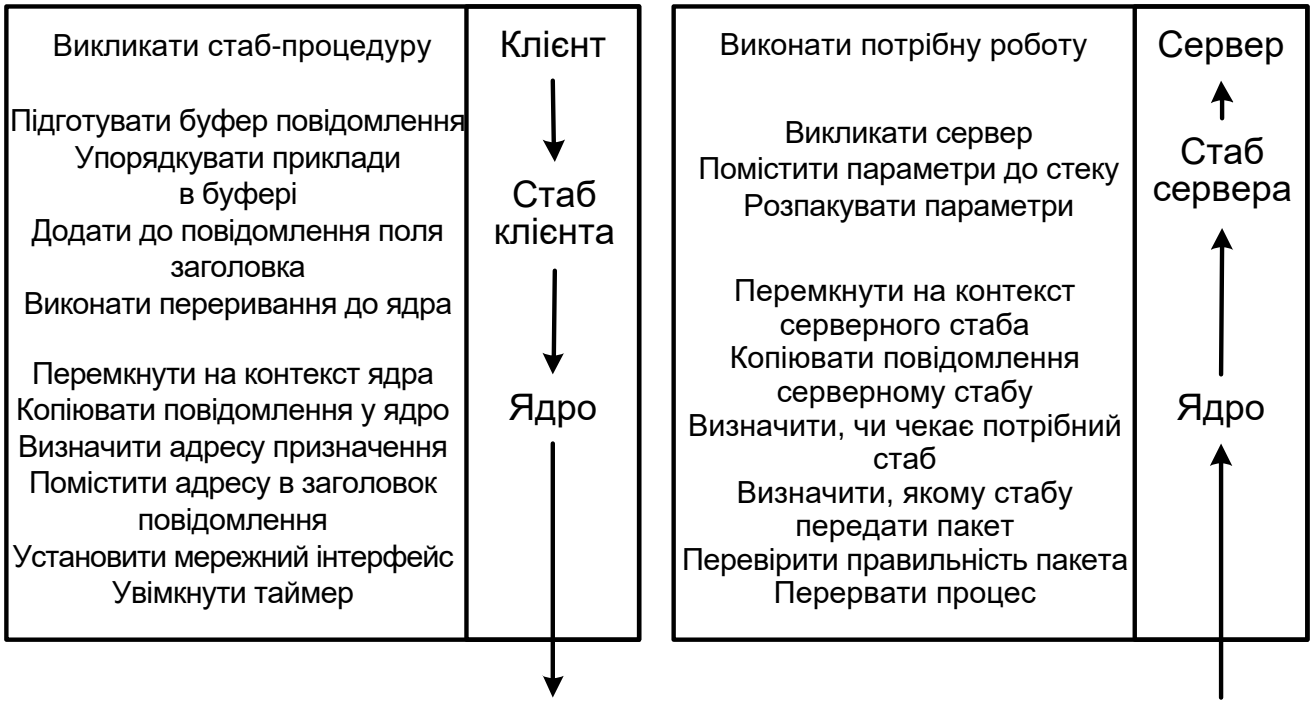

Рис. 4.2. Етапи виконання процедури RPC

Оброблювач переривання перевіряє правильність даних пакета й визначає, якому стабу слід їх передати. Якщо жоден зі стабів не очікує на цей пакет, то оброблювач повинен або помістити його в буфер, або взагалі відмовитися від нього. Якщо є стаб, що очікує, то повідомлення копіюється йому. Нарешті, виконується ***перемикання контекстів***, унаслідок чого відновлюються регістри й карта пам'яті, набуваючи тих значень, які вони мали в момент, коли стаб зробив виклик.

Тепер починає роботу ***серверний стаб***. Він розпаковує параметри й поміщає їх відповідним чином у стек. Після завершення роботи виконується виклик сервера.

Після виконання процедури сервер передає результати клієнту. Для цього виконуються всі описані вище етапи, тільки в зворотному порядку.

4.2. Реалізації RPC

Різні реалізації RPC мають архітектури, які дуже відрізняються одна від одної, різняться і їх можливості: одні реалізують архітектуру **SOA**, інші – **CORBA** або **DCOM**.

Існує безліч **технологій, що забезпечують RPC**, наприклад [24]:

– **DCE / RPC** – Distributed Computing Environment / Remote Procedure Calls (бінарний протокол на базі різних транспортних протоколів, у тому числі TCP / IP);

– **DCOM** – Distributed Component Object Model, відома як MSRPC Microsoft Remote Procedure Call або «Network OLE» (об'єктно-орієнтоване

розширення DCE/RPC, що дає змогу передавати посилання на об'єкти й викликати методи об'єктів через ці посилання);

– **JSON-RPC** – JavaScript Object Notation Remote Procedure Calls – протокол, який використовує JSON (текстовий формат обміну даними, що базується на JavaScript) для кодування повідомлень; JSON-RPC визначає кілька типів даних і команд, підтримує повідомлення і множинні виклики;

– **.NET Remoting** – створений компанією Microsoft компонент, що надає API для взаємодії між процесами; є частиною пакета .NET Framework 1.0 і фактично являє собою Microsoft-реалізацію протоколу SOAP;

– **Java RMI** – Java Remote Method Invocation – програмний інтерфейс виклику віддалених методів у мові Java, що являє собою розподілену об'єктну модель, специфікує способи викликів віддалених методів, які працюють на інший віртуальній машині Java;

– **XML RPC** – XML-виклик віддалених процедур – протокол, який використовує XML для кодування своїх повідомлень і HTTP як транспортний механізм і є прабатьком SOAP;

– **SOAP** – протокол обміну структурованими повідомленнями в розподіленому обчислювальному середовищі; спочатку SOAP призначався в основному для реалізації RPC, але зараз використовується для обміну довільними повідомленнями в форматі XML, а не тільки для виклику процедур;

– **Internet Communications Engine (Ice)** – розроблена ZeroC об'єктна система проміжного шару, яка використовує механізм RPC і поширюється під подвійною ліцензією: вільне ПЗ (GNU GPL) або комерційне; Ice підтримує багато платформ програмування, включаючи C++, Java, .NET, Visual Basic, Python, Ruby і PHP, і успішно конкурує з SOAP, оскільки використовує бінарний протокол передання даних, забезпечуючи менше навантаження на мережу й процесор.

4.3. Організація зв'язку з використанням віддалених об'єктів

Об'єктно-орієнтована технологія сьогодні широко застосовується під час розроблення додатків, у тому числі й розподілених. Однією з найбільш важливих властивостей об'єкта є те, що він приховує особливості реалізації, надаючи для взаємодії строго описаний інтерфейс. Це дає змогу замінювати або змінювати об'єкти, залишаючи інтерфейс незмінним.

Розвиток клієнт-серверної архітектури на початку 90-х років сприяв формуванню об'єктно-орієнтованої концепції розподілених систем, спрямованої на інкапсуляцію механізму розподілених взаємодій і спрощення розроблення розподілених додатків з допомогою методів об'єктно-орієнтованого розроблення й віддалених викликів методів об'єктів.

Переваги цього підходу:

- спрощення розроблення розподілених додатків порівняно з класичним клієнт-серверним підходом;

- можливість розроблення додатків для гетерогенних обчислювальних середовищ (забезпечується застосуванням віртуальних машин, наприклад Java, і незалежним описом інтерфейсів взаємодійних компонентів);

- можливість відділення інтерфейсу віддаленого об'єкта від його безпосередньої реалізації.

Віддалений об'єкт є деякими даними, сукупність яких визначає його ***стан***. Цей стан можна змінювати шляхом виклику ***методів*** об'єкта. Якщо можна безпосередньо отримати доступ до даних віддаленого об'єкта, то це здійснюється з допомогою неявного віддаленого виклику, необхідного для передання значення поля даних об'єкта між процесами. Методи й поля об'єкта, які можуть використовуватися через віддалені виклики, є доступними через деякий ***зовнішній інтерфейс*** класу об'єкта.

Для передання параметрів по мережі використовується серіалізація об'єктів і даних.

Серіалізация – це переведення стану об'єкта в послідовність бітів (найчастіше – бінарний або XML-файл), після чого його копія може бути передана в інший процес.

Зворотний процес – **десеріалізація** – це відновлення стану об'єкта з прийнятої послідовності бітів.

Схему організації зв'язку з використанням віддалених об'єктів зображено на рис. 4.3. У момент, коли клієнт починає використовувати віддалений об'єкт, на стороні клієнта створюється клієнтська заглушка, яка називається ***посередником*** (англ. *proxy*). Посередник реалізує той самий інтерфейс, що й віддалений об'єкт.

Процес, що викликає, використовує методи посередника, який серіалізує їх параметри для передання по мережі, і передає їх мережею серверу. Проміжне середовище на стороні сервера десеріалізує параметри й передає їх заглушці на стороні сервера, яку називають ***каркасом*** (англ. *skeleton*), або, як і у віддаленому виклику процедур, ***заглушкою***. Каркас зв'язується з деяким екземпляром віддаленого об'єкта. Це може бути як новостворений, так і існуючий екземпляр об'єкта залежно від моделі використання віддалених об'єктів, що застосовується.

При використанні віддалених об'єктів виникають проблемні запитання, пов'язані з часом їх життя:

1. У який момент часу створюється екземпляр віддаленого об'єкта?

2. Протягом якого проміжку часу він існує?

Для опису життєвого циклу в системах з віддаленими об'єктами використовують два додаткових поняття:

- активація об'єкта – процес переведення створеного об'єкта в стан обслуговування віддаленого виклику, тобто зв'язування з каркасом і посередником;

- деактивація об'єкта – процес переведення об'єкта в стан невикористання.

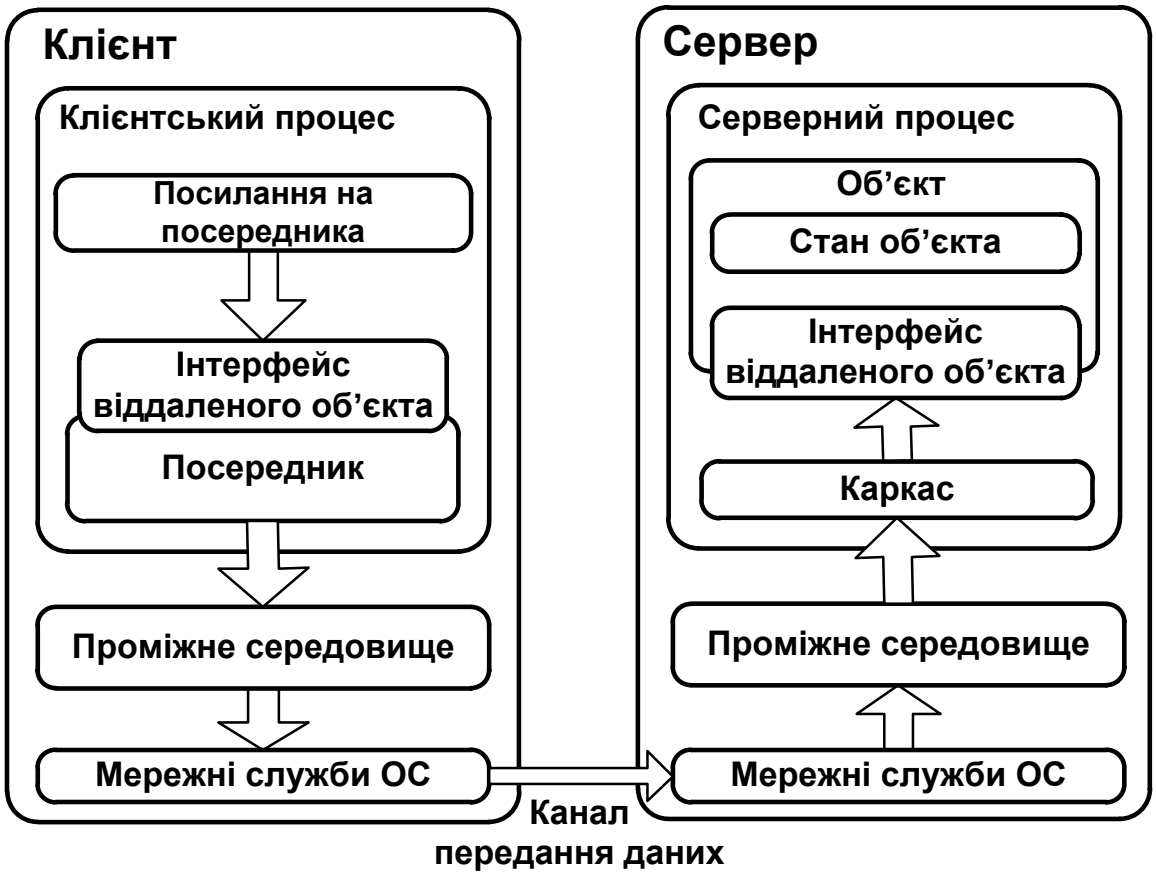

Рис. 4.3. Організація зв'язку з використанням віддалених об'єктів

Найбільш відомими прикладами реалізації систем із застосуванням віддалених об'єктів є Java RMI й архітектура CORBA [5].

4.4. Java RMI

Технологія ***Java RMI*** (Remote Method Invocation – виклик віддалених методів) дає змогу забезпечити прозорий доступ до методів віддалених об'єктів (ВО) шляхом доставлення параметрів методу, що викликається, повідомлення об'єкту про необхідність виконання методу й передання значення, що повертається клієнту.

Розподілений додаток, розроблений на базі технології Java RMI, складається з двох окремих програм: ***клієнта*** й ***сервера*** (рис. 4.4). Серверний додаток створює віддалений об'єкт, публікує посилання на нього й чекає, коли клієнти зроблять виклик методу цього віддаленого об'єкта. Додаток-клієнт отримує з сервера посилання на віддалений об'єкт на сервері, після чого може викликати його методи. Технологія RMI забезпечує механізм, з допомогою якого проводиться обмін інформацією між клієнтом і сервером.

Процес публікації посилання на віддалений об'єкт може бути реалізований з допомогою спеціального регістра або ж з допомогою передання вилученого об'єктного посилання як частини звичайної операції.

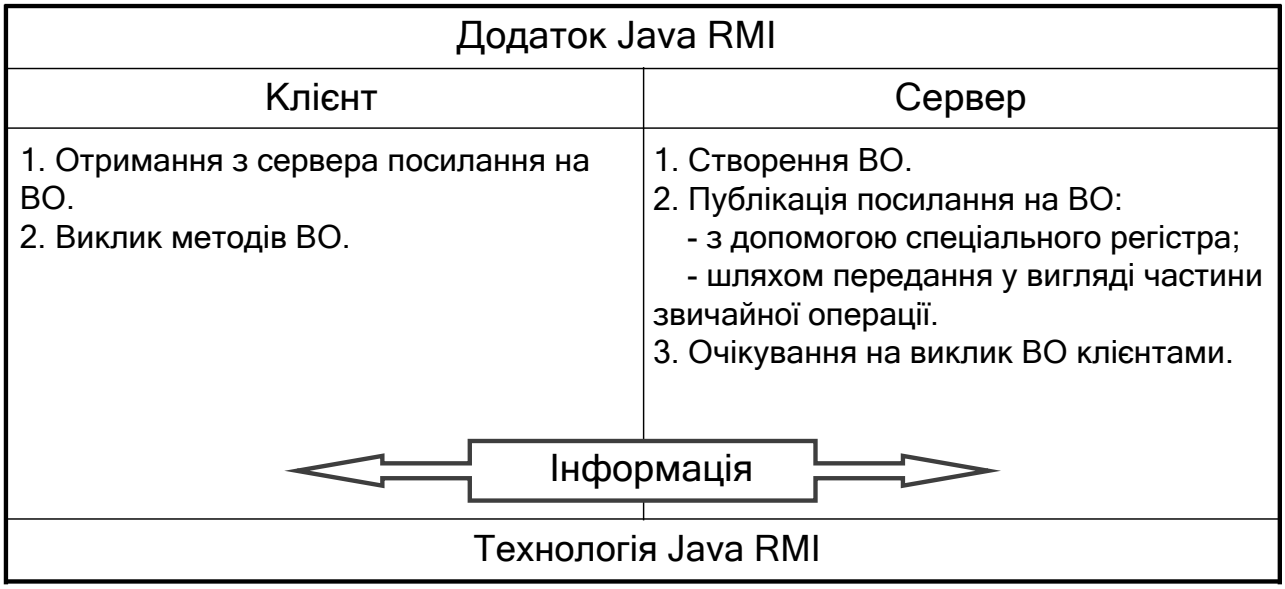

Рис. 4.4. Додаток Java RMI

Перевагами використання технології Java RMI для розроблення розподіленого додатка можна назвати можливість розробляти систему, цілком ґрунтуючись на об'єктно-орієнтованій концепції, не заглиблюючись у розроблення власних протоколів взаємодії між розподіленими компонентами систем, і кросплатформність, що надається віртуальною машиною Java. До недоліків цього підходу можна віднести:

- сувору обмеженість цієї технології платформою Java;
- обмежену масштабованість через необхідність оброблення з'єднань між розподіленими компонентами програми.

4.5. CORBA

CORBA (Common Object Request Broker Architecture – загальна архітектура брокера об'єктних запитів) – це технологія розроблення розподілених додатків, орієнтована на інтеграцію розподілених ізольованих систем.

Наприкінці 1980-х існувала серйозна ***проблема забезпечення взаємодії програм***, які виконуються на різних машинах, оскільки на цих машинах могли бути використані різні апаратні засоби, системи й мови програмування. Для вирішення цієї проблеми 1989 року створено консорціум **OMG (Object Management Group)**, основним завданням якого стало розроблення й просування об'єктно-орієнтованих технологій і стандартів. Це некомерційне об'єднання, що розробляє стандарти для створення корпоративних платформонезалежних додатків.

Концептуальною інфраструктурою, на якій базуються всі специфікації OMG, є **Object Management Architecture (OMA)**. До складу OMA входять різноманітні стандартизовані або такі, що стандартизуються зараз, OMG-сервіси, програмні зразки і шаблони, мова визначення інтерфейсів розподілених об'єктів **IDL (Interface Definition Language)**, стандартизовані або такі, що стандартизуються зараз, відображення IDL мовою програмування і, нарешті, об'єктна модель **CORBA**. Головною особливістю CORBA є використання компонента **ORB** (Object Resource Broker – брокер ресурсів об'єктів) для створення екземплярів об'єктів і виклику їх методів.

1997 року консорціум OMG опублікував специфікацію CORBA 2.0, у якій було визначено стандартний протокол і відображення для мови C++, а 1998 року було додано відображення для мови Java. Унаслідок цього розробники отримали інструментальний засіб, що дає змогу відносно легко створювати неоднорідні розподілені додатки. CORBA швидко набула популярності, і з використанням цієї технології було створено багато критично важливих додатків.

4.5.1. Технологія CORBA

Основні компоненти, з яких складається архітектура CORBA, показано на рис. 4.5. Технологічний стандарт CORBA визначає мову **IDL**, яка застосовується для уніфікованого опису інтерфейсів розподілених об'єктів і його відображення мовами Ada, C, C++, Java, Python, COBOL, Lisp, PL/1 і Smalltalk.

Для перетворення опису інтерфейсу мовою IDL на потрібну мову програмування використовується спеціальний ***компілятор***. Програмний код, побудований з його допомогою, можна перетворити будь-яким стандартним компілятором на виконуваний код.

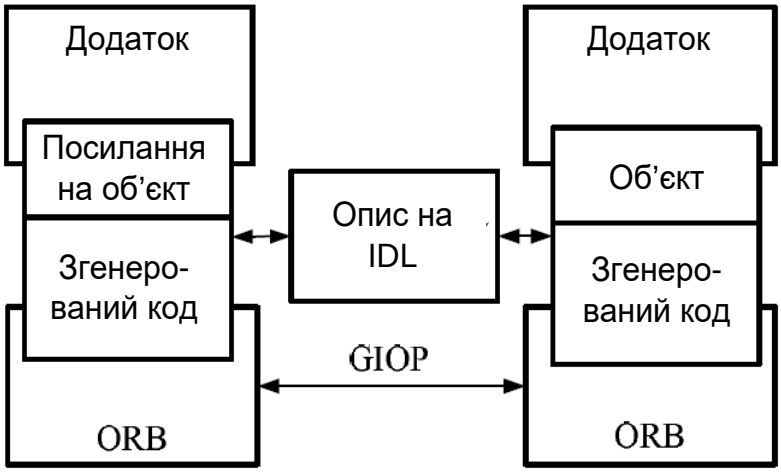

Рис. 4.5. Ядро архітектури CORBA

Для створення екземплярів об'єктів і виклику їх методів використовується компонент ORB, який формує «міст» між додатком та інфраструктурою CORBA. ORB підтримує віддалену взаємодію з іншими ORB, а також забезпечує керування ВО, включаючи врахування кількості посилань і часу життя об'єкта.

Для забезпечення взаємодії між ORB використовується протокол **GIOP** (**General Inter-ORB Protocol** – *загальний протокол для комунікації*

між ORB). Найбільш поширеною реалізацією цього протоколу є протокол **IIOP** (**Internet Inter-ORB Protocol** – *протокол взаємодії ORB у мережі інтернет*), що забезпечує відображення повідомлень GIOP на стек протоколів TCP/IP.

Спочатку технологію CORBA було створено з метою надання готової проблемно-орієнтованої інфраструктури для створення РОС у межах певної проблемної області. Тому до складу CORBA входять такі набори (рис. 4.6):

1. ***Стандартні об'єктні сервіси*** (*CORBA Services*):

- NameService – сервіс імен;

- сервіс повідомлень, що дає змогу CORBA-об'єктам обмінюватися повідомленнями;

- сервіс транзакцій, що дає змогу CORBA-об'єктам організовувати транзакції тощо.

2. ***Спільні засоби*** (*Common Facilities*), які, своєю чергою, поділяються на такі:

- горизонтальні – для всіх прикладних областей;

- вертикальні – для конкретної прикладної області (наприклад, для медичних організацій, виробничих сфер та ін.).

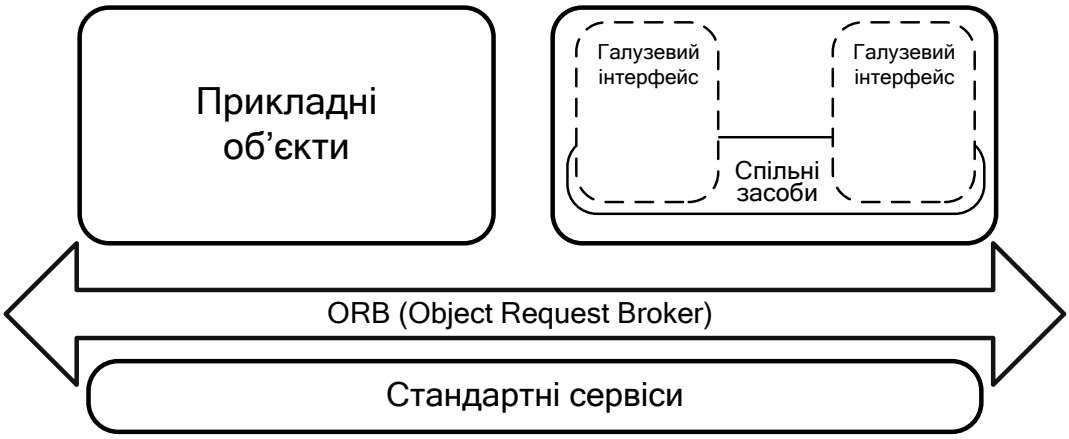

Рис. 4.6. Склад архітектури CORBA

Між об'єктними сервісами і спільними засобами CORBA немає чіткої межі. І одні, і інші являють собою CORBA-об'єкти зі стандартизованими інтерфейсами. Слід зазначити, що в реальній системі не обов'язково мають бути всі сервіси, їх набір залежить від необхідної функціональності. На сьогодні розроблено 14 об'єктних сервісів.

4.5.2. Розроблення додатків на основі CORBA

Процес розроблення додатка з використанням технології CORBA складається з чотирьох етапів:

1. Визначення інтерфейсу мовою IDL.

2. Оброблення IDL для створення коду заглушки і скелетона.

3. Створення коду реалізації об'єкта (на стороні сервера).

4. Створення коду використання об'єкта (на стороні клієнта).

Мова визначення IDL дає змогу незалежно від мови програмування створити універсальний опис інтерфейсу майбутньої системи. Код, створений мовою IDL, має перетворюватися спеціальним компілятором на код інтерфейсу об'єкта необхідною мовою програмування. Після цього на клієнті автоматично генерується заглушка, що перетворює виклики методів цього інтерфейсу на звертання до ORB. На сервері програміст на основі згенерованого інтерфейсу створює власну реалізацію цього класу. Скелетон автоматизує отримання й оброблення віддаленого виклику методів, що надходять через ORB.

Порівняно з класичним клієнт-серверним підходом використання технології CORBA для розроблення розподілених додатків має такі переваги:

- можливість розроблення програмних компонентів незалежно від мови програмування й базової операційної системи при використанні IDL для опису інтерфейсів;
- підтримка розвиненої інфраструктури розподілених об'єктів;
- прозорість виклику віддалених об'єктів.

Однак програмні рішення на базі технології CORBA рідко виходять за межі окремих підприємств. Розроблення великомасштабних міжустановних систем на базі технології CORBA пов'язане з такими труднощами:

- погана сумісність різних реалізацій технології CORBA від різних постачальників;
- проблеми взаємодії вузлів CORBA через інтернет;
- неузгодженість багатьох архітектурних рішень CORBA і відсутність компонентної моделі, яка могла б значно спростити розроблення.

Технологію CORBA замінили стандартизовані протоколи веб-сервісів, такі як XML, WSDL, SOAP та ін. Нині CORBA використовується для реалізації вузького кола успадкованих додатків.

5. КОМПОНЕНТНІ СИСТЕМИ

Компонентно-орієнтований підхід (КОП) до проектування й реалізації програмних систем і комплексів в деякому сенсі являє собою ***розвиток об'єктно-орієнтованого підходу*** і практично є більш придатним для розроблення великих і розподілених програмних систем (наприклад, корпоративних додатків).

З огляду на КОП **програмна система** – це набір компонентів з чітко визначеним інтерфейсом. На відміну від інших підходів програмної інженерії зміни в систему вносяться шляхом створення нових компонентів або змінення старих, а не шляхом рефакторингу існуючого коду.

Програмний компонент – це автономний елемент програмного забезпечення, призначений для багаторазового використання, який може застосовуватися в інших програмах у вигляді скомпільованого коду. Підключення до програмних компонентів здійснюється з допомогою відкритих інтерфейсів, а взаємодія з програмним середовищем – щодо подій, причому в програмі, що використовує компонент, можна призначати обробників подій, на які вміє реагувати компонент.

Застосування компонентного програмування забезпечує більш просту, швидку й прямолінійну процедуру початкової інсталяції прикладного ПЗ, а також збільшує відсоток повторного використання коду, тобто підсилює основні переваги ООП.

Властивості компонентів:

- істотно більший розмір порівняно з об'єктами;
- можливість утримувати множинні класи;
- незалежність від мови програмування.

Слід зазначити, що автор і користувач компонента є територіально розподіленими й можуть не тільки писати програми, а й говорити різними мовами.

Можна виділити такі **основні переваги застосування КОП** при проектуванні й розробленні ПЗ:

- зниження вартості програмного забезпечення;
- підвищення повторного використання коду;
- уніфікація оброблення об'єктів різної природи;
- менш людинозалежний процес створення ПЗ.

Оскільки програмний компонент (ПК) передбачає повноцінне автономне використання у вигляді «чорного ящика», до розроблення ПК ставляться жорсткі вимоги:

- ***повна документованість інтерфейсу**:* усі методи, що надаються в інтерфейсі ПК, мають бути якісно задокументованими з урахуванням усіх можливих варіантів їх використання в сторонніх додатках;
- ***ретельне тестування**:* необхідно враховувати всі можливі й неможливі варіанти використання ПК у сторонніх системах на всіх можливих значеннях вхідних даних;
- ***ретельний аналіз вхідних значень**:* слід ураховувати можливість передання в ПК вхідних даних, які не відповідають його специфікації, та адекватно обробляти такі ситуації;
- ***повернення адекватних і зрозумілих повідомлень про помилки**:* оскільки один ПК може бути використаний у багатьох сторонніх програмних системах, необхідно забезпечити стороннім розробникам можливість отримувати інформації про помилки ПК і варіанти їх вирішення;
- ***необхідність передбачити можливість неправильного використання.***

Найбільш відомими прикладами компонентних систем є технології COM [25] і EJB [5, 26].

5.1. Component Object Model

COM (**Component Object Model** – *об'єктна модель компонентів*) – це технологічний стандарт від компанії Microsoft, призначений для створення ПЗ на основі взаємодійних компонентів, кожен з яких може використовуватися в багатьох програмах одночасно. Стандарт втілює в собі ідеї поліморфізму й інкапсуляції об'єктно-орієнтованого програмування.

Стандарт COM був розроблений 1993 року корпорацією Microsoft як основа для розвитку технології **OLE** (**Object Linking and Embedding** – *зв'язування і вбудовування об'єкта*). Технологія OLE 1.0, що існувала на той момент, уже давала змогу створювати так звані складені документи (*compound documents*) (наприклад, у пакеті Microsoft Office включати діаграми Microsoft Excel у документи Microsoft Word).

Стандарт COM міг би бути універсальним і платформонезалежним, але закріпився в основному на ОС сімейства Microsoft Windows. На основі COM реалізовано технології Microsoft OLE Automation, ActiveX, DCOM, COM+, DirectX, а також XPCOM.

5.1.1. Принципи роботи COM

Основним поняттям, яким оперує стандарт COM, є COM-компонент. Програми, побудовані на стандарті COM, фактично не є автономними програмами, а являють собою набір взаємодійних COM-компонентів.

Кожен компонент має унікальний ідентифікатор (**GUID – Globally Unique Identifier**) і може одночасно використовуватися багатьма програмами.

Компонент взаємодіє з іншими програмами через **COM-інтерфейси** – набори абстрактних функцій і властивостей. Кожен COM-компонент повинен щонайменше підтримувати стандартний інтерфейс **IUnknown**, який надає базові засоби для роботи з компонентом і **містить три методи**:

- *QueryInterface,* призначений для отримання покажчика на інтерфейс COM-компонента;
- *AddRef,* що збільшує кількість посилань на певний COM-компонент на одиницю;
- *Release*, що зменшує кількість посилань на певний COM-компонент на одиницю; якщо при черговому виклику Release кількість посилань дорівнює нулю, то COM-компонент має знищитися і звільнити всі ресурси.

Разом функції *AddRef* і *Release* застосовуються для керування часом життя COM-компонента, що експортує інтерфейси.

Базові функції, що дають змогу використовувати COM-компоненти, надає **Windows API**.

Windows API, який спроектовано для використання в мові **C** для написання прикладних програм, призначених для роботи під керуванням операційної системи MS Windows, являє собою безліч функцій, структур

даних і числових констант, які відповідають угодам мови С. Усі мови програмування, здатні викликати такі функції й оперувати такими типами даних у програмах, що виконуються в середовищі Windows, можуть користуватися цим API. Зокрема, це мови **C++, Pascal, Visual Basic** і багато інших.

Існують бібліотеки й середовища програмування, що надають ту або іншу частину Windows API у більш зручному вигляді, наприклад, розроблені Microsoft: **MFC, ATL/WTL, .Net/WinForms/WPF**.

Microsoft Foundation Classes (MFC) – бібліотека мовою C++ для полегшення розроблення GUI-додатків для Microsoft Windows шляхом використання широкого набору бібліотечних класів. MFC генерує каркас додатка – «кісткову» програму, що автоматично створюється за заданим макетом інтерфейсу й повністю виконує рутинні дії з обслуговування додатка (відпрацювання віконних подій, пересилання даних між внутрішніми буферами елементів і змінними тощо). Програмісту після генерації каркаса додатка необхідно тільки вписати код у місця, де потребуються спеціальні дії.

Active Template Library (ATL) – набір шаблонних класів мови C++, розроблених для спрощення написання COM-компонентів. Ця бібліотека дає змогу розробникам створювати різні об'єкти COM, сервери автоматизації OLE і керувальні елементи ActiveX. До складу середовища розроблення Visual Studio входять майстер і помічники для ATL, що дає змогу створювати первинну об'єктну структуру практично без програмування вручну. ATL – це деякою мірою полегшена альтернатива MFC як засіб керування COM.

Windows Template Library (WTL) – вільно поширювана бібліотека шаблонів (шаблонних класів) C++, призначена для стандартних GUI-додатків Windows, що є розширенням бібліотеки ATL. WTL є надбудовою над інтерфейсом Win32 API, її розробляли насамперед як полегшену альтернативу бібліотеки MFC. WTL підтримує роботу з вікнами й діалогами, стандартними діалогами Windows, GDI (Graphics Device Interface – інтерфейс Windows для зображення графічних об'єктів і передання їх на пристрої відображення (наприклад, монітори, принтери)), стандартними контролами, ActiveX та ін. У бібліотеці подано основні елементи керування: меню, панелі інструментів, кнопки, поля введення, списки і т. д.

5.1.2. Технології, що базуються на стандарті COM

На стандарті COM базуються багато технологій, основними серед яких є технології **OLE** [25, 27], **COM+** [28, 29] і **DCOM** [25, 27].

OLE (**Object Linking and Embedding** – зв'язування і вбудовування об'єкта) – технологія зв'язування об'єктів та їх впровадження в інші документи й об'єкти. OLE дає змогу передавати частину роботи від однієї програми редагування до іншої й повертати результати назад. Наприклад, установлена на персональному комп'ютері видавнича система може послати якийсь текст на оброблення в текстовий редактор або деяке зобра-

ження в редактор зображень з допомогою OLE-технології.

1996 року Microsoft спробувала перейменувати OLE в **ActiveX**, але це вдалося лише частково й призвело до плутанини в термінах. Терміном ActiveX зараз називають усе, що належить до OLE, плюс деякі нововведення, проте згоди з питання про точне означення ActiveX серед експертів з DCOM (навіть усередині Microsoft) не існує. Загальну структуру технологій COM, OLE і ActiveX показано на рис. 5.1 [27].

Рис. 5.1. Загальна структура ActiveX, OLE і COM

Технологія **COM+** (раніше – **Microsoft Transaction Server, MTS**), призначена для підтримки систем оброблення транзакцій, базується на можливостях COM і забезпечує підтримку розподілених додатків на компонентній основі. Об'єкти транзакцій COM+ мають основні властивості об'єктів COM. Крім того, об'єкти транзакцій реалізують **специфічні можливості**: керування транзакціями, безпека, пулінг ресурсів та об'єктів.

Особливості COM+ об'єктів:

- реалізація в складі внутрішнього сервера (динамічна бібліотека);
- наявність посилання на бібліотеку типів COM+;
- використання тільки стандартного механізму ***маршалінгу*** (запис стану й кодових баз об'єкта таким чином, щоб при зворотній операції виходила копія оригіналу) COM;
- імплементація інтерфейсу IobjectControl.

У COM+ можливе використання **двох типів об'єктів**:

- statefull (зі збереженням інформації про стан об'єкта);
- stateless (без збереження інформації про стан об'єкта).

Принцип роботи COM+ полягає в такому. COM+ – це сукупність програмних засобів, що забезпечують розроблення, поширення й

функціонування розподілених додатків для мереж інтернет та інтранет. До її **складу** входять:

- ПЗ проміжного рівня, що забезпечує функціонування об'єктів транзакцій під час виконання завдання;
- утиліта MTS Explorer, що дає змогу керувати об'єктами транзакцій;
- інтерфейси прикладного програмування;
- засоби керування ресурсами.

Стандартна програмна модель додатків, що використовують COM+, являє собою триланкову архітектуру розподілених додатків. При цьому бізнес-логіку програми сконцентровано в **об'єктах транзакцій**, а ПЗ проміжного рівня, що керує цими об'єктами, побудовано з використанням компонентної моделі. Розробники, що використовують COM+ у своїх додатках, спочатку створюють об'єкти бізнес-логіки, що задовольняють вимогам до об'єктів COM+, а потім компілюють їх і встановлюють в середовищі COM + за допомогою пакетів. Пакет COM + являє собою контейнер, що забезпечує групування об'єктів з метою захисту даних, поліпшення керування ресурсами і підвищення продуктивності. Керування пакетами здійснюється з допомогою утиліти MTS Explorer.

DCOM (Distributed COM) – розширення COM для підтримки зв'язку між розподіленими (віддаленими) об'єктами. Загальну архітектуру DCOM зображено на рис. 5.2.

Для створення об'єкта на віддаленій машині бібліотека COM викликає **SCM** (менеджер керування сервісами) локального комп'ютера, який зв'язується з **SCM** сервера й передає йому запит на створення об'єкта. Ім'я сервера може задаватися при виконанні функції створення об'єкта або зберігатися в реєстрі.

Для виклику віддаленого об'єкта параметри мають бути вилучені зі ***стека*** (або з ***регістрів процесора***), поміщені в буфер і передані через мережу. Процес вилучення параметрів і розміщення їх в буфер називають **маршалінгом**. Цей процес – нетривіальний, оскільки параметри можуть містити покажчики на масиви і структури, які, своєю чергою, можуть містити покажчики на інші структури. На сервері здійснюється зворотний процес відтворення стеку (**демаршалінг**), після чого викликається потрібний об'єкт. Після завершення виклику виконується маршалінг значення, що повертається, і вихідних параметрів і відправлення їх клієнту.

Для виконання маршалінгу і демаршалінгу необхідно мати точний опис методу, включаючи всі типи даних і розміри масивів. Для цього використовується мова опису інтерфейсів (IDL), що входить до стандарту DCE RPC. Отримані файли опису компілюються спеціальним компілятором IDL у вихідний мовою C, що виконує маршалінг і демаршалінг для зазначених інтерфейсів. Код, що запускається на стороні клієнта, має назву «проксі», а на стороні об'єкта – «стаб» і завантажується бібліотекою COM, якщо потрібно.

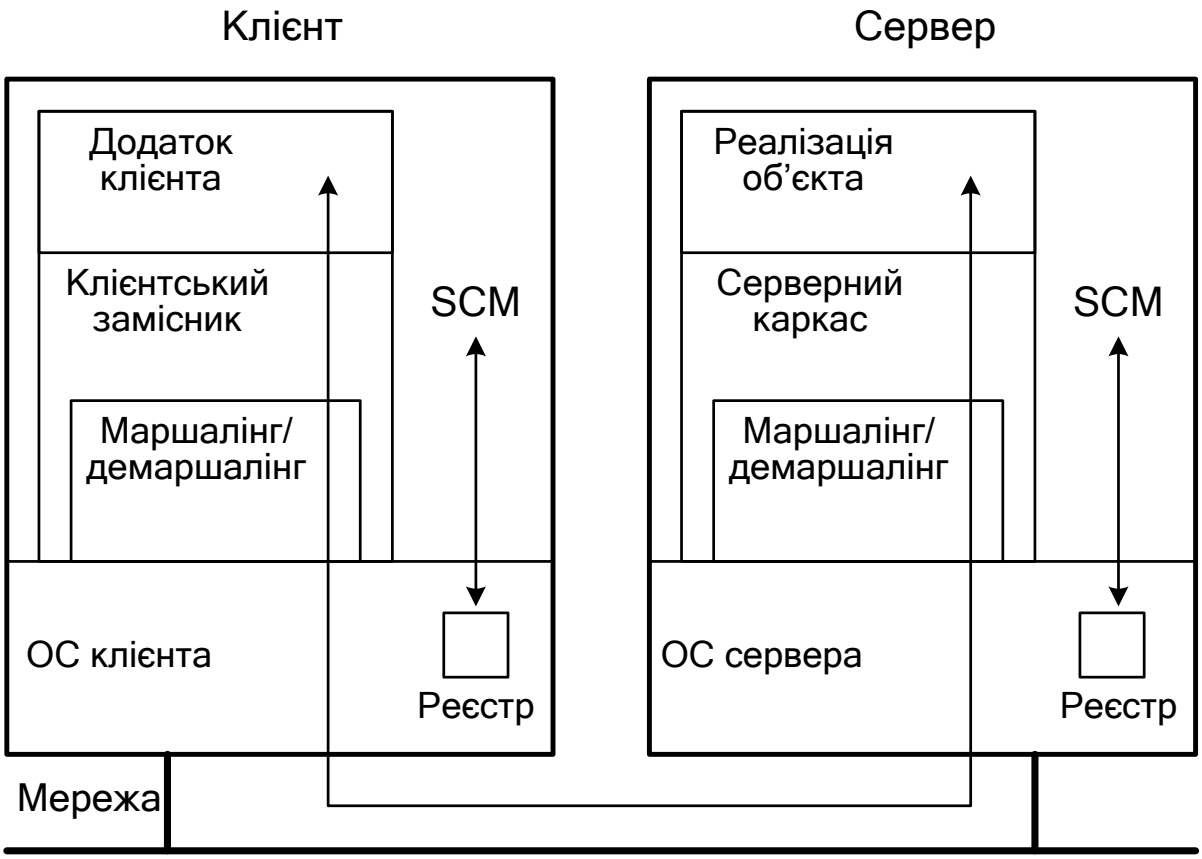

Рис. 5.2. Загальна архітектура DCOM

5.2. Концепція JavaBeans

JavaBeans – класи в мові Java, написані за певними правилами, що використовуються для об'єднання декількох об'єктів в один (bean) для зручного передання даних. JavaBeans забезпечують основу для багаторазово використовуваних, вбудованих і модульних компонентів ПЗ.

У специфікації Sun Microsystems JavaBeans визначаються як *універсальні програмні компоненти, якими можна керувати з допомогою графічного інтерфейсу*.

Компоненти JavaBeans можуть набирати різних форм, але найбільш широко вони застосовуються в елементах графічного інтерфейсу. Одна з цілей створення JavaBeans – *взаємодія зі схожими компонентними структурами*. Наприклад, Windows-програма за наявності відповідного моста або об'єкта-обгортки може використовувати компонент JavaBeans так, нібито він є компонентом COM або ActiveX.

Щоб клас міг працювати як bean, він має відповідати певним домовленостям про імена методів, конструктор і поведінку. Ці домовленості дають можливість створювати інструменти, які можуть використовувати, заміщати і з'єднувати JavaBeans.

Правила опису класу:

1. ***Клас повинен мати public-конструктор без параметрів.*** Такий конструктор дає змогу інструментам створювати об'єкт без додаткових складнощів з параметрами.

2. ***Властивості класу мають бути доступними через get, set та інші методи доступу,*** які підпорядковуються стандартній домовленості про імена. Це дає можливість інструментам автоматично визначати й оновлювати зміст bean. Багато інструментів мають також спеціалізовані редактори для різних типів властивостей.

3. ***Клас має підтримувати можливість серіалізації.*** Це дає можливість надійно зберігати й відновлювати стан bean способом, що не залежить від платформи і віртуальної машини.

4. ***Клас не має містити ніяких методів оброблення подій.***

5.2.1. Enterprise JavaBeans

Enterprise JavaBeans – це високорівнева технологія створення розподілених додатків, що базується на використанні компонентів і використовує низькорівневий API для керування транзакціями. Перший варіант специфікації Enterprise JavaBeans створено в березні 1998 р. За час свого існування технологія пройшла великий шлях і продовжує розвиватися.

Enterprise JavaBeans – більше, ніж просто інфраструктура. Її використання передбачає ще й технологію (процес) створення розподіленого додатка, нав'язує певну архітектуру програми, а також визначає стандартні ролі для учасників розроблення.

Застосування цих технік забезпечує розв’язання таких **стандартних проблем** масштабованих і ефективних **серверів додатків з використанням Java**:

- організація віддалених викликів між об'єктами, що працюють під керуванням різних віртуальних машин Java;
- керування потоками на стороні сервера;
- керування циклом життя серверних об'єктів (створення, взаємодія з користувачем, знищення);
- оптимізація використання ресурсів (часу процесора, пам'яті, мережних ресурсів);
- створення схеми взаємодії контейнерів та операційних середовищ;
- побудова схеми взаємодії контейнерів і клієнтів, у тому числі універсальних засобів розроблення компонентів і включення їх до складу контейнерів;
- розроблення засобів адміністрування й забезпечення їх взаємодії з наявними системами;
- створення універсальної системи пошуку клієнтом необхідних серверних компонентів;
- забезпечення універсальної схеми керування транзакціями;
- забезпечення необхідних прав доступу до серверних компонентів;

- забезпечення універсальної взаємодії з СКБД.

Технологія Enterprise JavaBeans визначає набір універсальних компонентів, призначених для багаторазового використання, – Enterprise beans (компонентів EJB). При створенні розподіленої системи її бізнес-логіка буде реалізована в цих компонентах.

Складові частини компонента EJB:

- ***віддалений інтерфейс*** (remote-інтерфейс), що визначає бізнес-методи, які може викликати клієнт EJB;
- ***власний інтерфейс*** (home-інтерфейс), який надає методи:
  - *create* для створення нових екземплярів компонентів EJB;
  - *finder* для знаходження екземплярів компонентів EJB;
  - *remove* для видалення екземплярів EJB;
- ***реалізація EJB-компонента,*** яка визначає бізнес-методи, оголошені у віддаленому інтерфейсі, і методи створення, видалення й пошуку власного інтерфейсу.

Після завершення розроблення набори компонентів EJB поміщаються в спеціальні файли (архіви, jar), по одному або більше компоненту, разом зі спеціальними *параметрами розгортання*. Потім вони встановлюються у спеціальному операційному середовищі, у якому запускається контейнер EJB. Клієнт здійснює пошук компонентів у контейнері з допомогою *home-інтерфейсу* відповідного компонента. Після того, як компонент створено і/або знайдено, клієнт виконує звернення до його методів з допомогою *remote-інтерфейсу*.

Контейнери EJB виконуються під керуванням ***сервера EJB*** (рис. 5.3), який є сполучною ланкою між контейнерами та операційним середовищем. Сервер EJB забезпечує доступ контейнерів EJB до системних сервісів, таких як керування доступом до баз даних або моніторинг транзакцій. Усі ***екземпляри компонентів EJB*** виконуються під керуванням контейнера EJB, який надає системні сервіси розміщеним у ньому компонентів і керує їх життєвим циклом.

У загальному випадку **контейнер призначено для вирішення таких завдань**:

1. ***Забезпечення безпеки.*** Дескриптор розгортання (deployment descriptor) визначає права доступу клієнтів до бізнес-методів компонентів. Забезпечення захисту даних здійснюється шляхом надання доступу тільки для авторизованих клієнтів і тільки до дозволених методів.

2. ***Забезпечення віддалених викликів.*** Контейнер виконує всі низькорівневі питання забезпечення взаємодії та організації віддалених викликів, повністю приховуючи всі деталі процесу як від розробника компонентів, так і від клієнтів. Це дає змогу розробляти компоненти так само, як і у випадку роботи системи в локальній конфігурації, тобто без використання віддалених викликів.

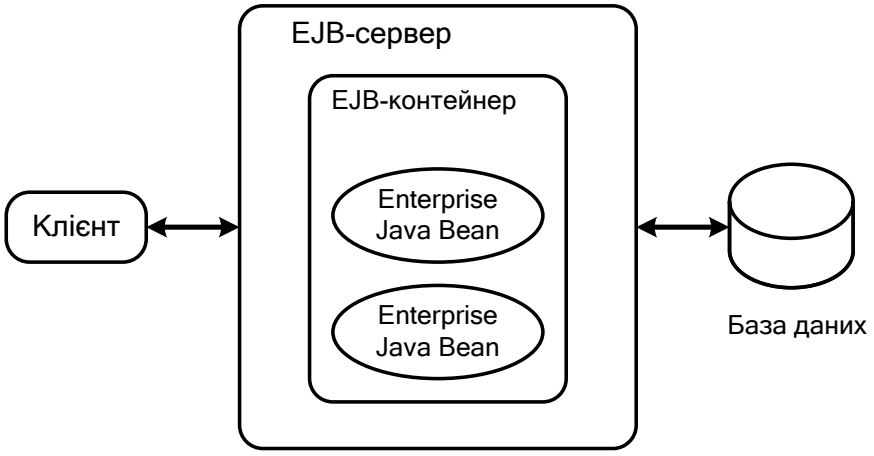

Рис. 5.3. Структура EJB-сервера

3. ***Керування життєвим циклом.*** Клієнт створює і знищує екземпляри компонентів, однак контейнер для оптимізації ресурсів і підвищення продуктивності системи може самостійно виконувати різні дії, наприклад активацію і деактивацію цих компонентів, створення їх пулів і т. д.

4. ***Керування транзакціями***. Усі параметри, необхідні для керування транзакціями, поміщаються в дескриптор поставки. Усі завдання щодо забезпечення керування розподіленими транзакціями в гетерогенних середовищах і взаємодії з декількома базами даних виконує контейнер EJB. Контейнер забезпечує захист даних і гарантує успішне підтвердження внесених змін; в іншому випадку транзакція відкочується.

5.2.2. Типи компонентів EJB

Існують *три типи компонентів EJB*:
- сесійні (Session Beans);
- сутнісні (Entity Beans);
- керовані повідомленнями (Message Driven Beans).

Сесійний компонент являє собою об'єкт, створений для обслуговування запитів одного клієнта. У відповідь на віддалений запит клієнта контейнер створює екземпляр такого компонента. Сесійний компонент завжди зіставляється з одним клієнтом, і його можна розглядати як «представника» клієнта на стороні EJB-сервера.

Сесійні компоненти є тимчасовими об'єктами. Зазвичай сесійний компонент існує доти, доки клієнт, що його створив, підтримує з ним сеанс зв'язку. Після завершення зв'язку з клієнтом компонент вже ніяк з ним не зіставляється.

Сесійні компоненти бувають трьох типів:
- stateless (без стану);
- stateful (з підтримкою поточного стану сесії);
- singleton (один об'єкт на весь додаток, починаючи з версії 3.1).

Сутнісні компоненти – це об'єктне подання даних з бази даних. Ключовою відмінністю сутнісного компонента від сесійного є те, що кілька клієнтів можуть одночасно звертатися до одного екземпляра сутнісного

компонента. Сутнісні компоненти змінюють стан зіставлених з ними баз даних у контексті транзакцій.

Стан компонентів-сутностей у загальному випадку потрібно зберігати, і живуть вони стільки, скільки існують в базі даних ті дані, які вони представляють, а не стільки, скільки існує клієнтський або серверний процес. Зупинка або крах контейнера EJB не призводить до знищення сутнісних компонентів, які в ньому містяться.

Керовані повідомленнями компоненти характеризуються тим, що їх логіка є реакцією на події в системі.

5.2.3. Складові частини EJB-компонента

EJB-компонент фізично складається з декількох частин, включаючи сам компонент, реалізацію деяких інтерфейсів та інформаційний файл. Усе це збирається разом у спеціальний **jar-файл – модуль розгортання**.

Enterprise Bean є Java-класом, розробленим постачальником Enterprise Bean, і забезпечує наявність інтерфейсу Enterprise Bean і реалізацію бізнес-методів, які виконує компонент. Клас не реалізує ніяких методів авторизації, багатопоточності або підтримки транзакцій.

Домашній інтерфейс. Кожен створюваний Enterprise Bean повинен мати асоційований домашній інтерфейс, що застосовується як фабрика для компонента EJB. Клієнт використовує домашній інтерфейс для знаходження екземпляра компонента EJB або створення нового екземпляра компонента EJB.

Віддалений інтерфейс є Java-інтерфейсом, що відображає через рефлексію ті методи Enterprise Bean, які необхідно показувати зовнішньому світу. Віддалений інтерфейс відіграє ту ж роль, що й IDL-інтерфейс у CORBA, і забезпечує можливість звернення клієнта до компонента.

Описувач розгортання є XML-файлом, який містить інформацію про компоненти EJB. Використання XML дає змогу установникові легко змінювати атрибути компонента. Конфігураційні атрибути, визначені в описувачі розгортання, містять:

- імена домашнього і віддаленого інтерфейсів;
- ім'я JNDI для публікації домашнього інтерфейсу компонента;
- транзакційні атрибути для кожного методу компонента;
- контрольний список доступу для авторизації.

EJB-Jar-файл – це звичайний java-jar-файл, який містить компонент (компоненти) EJB, домашній і віддалений інтерфейси, а також описувач розгортання.

Інфраструктура EJB забезпечує віддалену взаємодію об'єктів, керування транзакціями й безпеку додатків. Специфікація EJB обумовлює вимоги до елементів інфраструктури й визначає Java API, проте вона не стосується питань вибору платформ, протоколів та інших аспектів, пов'язаних з реалізацією.

У загальному випадку необхідно гарантувати збереження стану компонентів у контейнерах. Інфраструктура EJB має надати можливості для інтеграції додатка з наявними системами й додатками. Усі аспекти взаємодії клієнтів з серверними компонентами мають відбуватися в контексті транзакцій, керування якими покладається на інфраструктуру EJB.

Специфікація Enterprise JavaBeans – це суттєвий крок до стандартизації моделі розподілених об'єктів у Java.

6. СЕРВІС-ОРІЄНТОВАНА АРХІТЕКТУРА

Сервіс-орієнтована архітектура *(COA, Service-Oriented Architecture – SOA)* – це парадигма організації та використання розподілених можливостей, які можуть належати різним власникам.

COA – це модульний підхід до розроблення програмного забезпечення, що базується на використанні розподілених, слабо зв'язаних замінних компонентів, оснащених стандартизованими інтерфейсами для взаємодії за стандартизованими протоколами [5].

Архітектура не зв'язана з якоюсь певною технологією. Вона може бути реалізована з використанням широкого спектра технологій, включаючи такі технологи, як ***REST***, ***RPC***, ***DCOM***, ***CORBA*** або ***веб-сервіси***. SOA може використовувати один з цих протоколів, а також додатково механізм файлової системи для обміну даними.

Головне, що відрізняє SOA від інших архітектур, – це використання незалежних сервісів з чітко визначеними інтерфейсами, що для виконання їх завдань можуть бути викликані якимось стандартним способом за умови, що сервіси заздалегідь нічого не знають про програму, яка їх викличе, а додаток не знає, яким чином сервіси виконують своє завдання.

6.1. Складові COA

Типовими складовими COA є:

- *сервісні компоненти* (сервіси);
- *контракти сервісів* (інтерфейси);
- *з'єднувачі сервісів* (транспорт);
- *механізми виявлення сервісів* (регістри).

Сервісні компоненти (або сервіси) – це відкриті програмні компоненти, що самовизначаються й надають певної функціональності. Залежно від обсягу наданих послуг виділяють такі сервіси:

- *дрібномодульні (ДМС)*, що надають елементарний обсяг функціонального навантаження й забезпечують високий ступінь повторного використання; іноді для отримання бажаного результату необхідно забезпечити координовану роботу декількох ДМС;

- *крупномодульні (КМС),* що дають змогу забезпечити хорошу інкапсуляцію функціональності; повторне використання КМС є ускладненим че-

рез їх вузьку спеціалізацію.

Контракт сервісу (або інтерфейс) забезпечує опис можливостей і якості послуг, що надаються конкретним сервісом. В інтерфейсі визначається формат повідомлень, що використовується для обміну інформацією, а також вхідні й вихідні параметри методів, що підтримуються сервісним компонентом. Від вибору мови й способу опису інтерфейсу залежать можливості програмної сумісності різних реалізацій СОА.

З'єднувач сервісів (або транспорт) забезпечує обмін інформацією між окремими сервісними компонентами. Поряд з відкритими стандартами опису інтерфейсів використання гнучких транспортних протоколів для обміну інформацією між сервісними компонентами дає змогу підвищити програмну сумісність сервіс-орієнтованої системи.

Механізми пошуку послуг (або регістри сервісів) використовуються для пошуку сервісних компонентів, що забезпечують необхідну функціональність.

Виділяють дві основні категорії систем виявлення:

- *статичні системи* виявлення сервісів (наприклад, UDDI), орієнтовані на зберігання інформації про сервіси в системах, що рідко змінюються;
- *динамічні системи* виявлення сервісів, орієнтовані на системи, у яких допустимими є часті появи і зникнення сервісних компонентів.

6.2. Зв'язаність програмних систем

Зв'язаністю називають ступінь знання й залежності одного об'єкта від внутрішнього змісту іншого.

Програмні системи можна поділити на два типи:

- *сильнозв’язані системи (strong coupling):* залежний клас містить посилання безпосередньо на певний клас, який надає деякі можливості (приклади: Java RMI, NET Remoting);
- *слабозв'язані системи (loose coupling):* залежний клас містить посилання на інтерфейс, який може бути реалізований одним або декількома конкретними класами (наприклад, СОА).

Основна мета використання концепції слабозв’язаних програмних систем – це зменшення кількості залежностей між компонентами. При зменшенні кількості зв'язків зменшується обсяг можливих наслідків, що виникають через збої або системні змінення.

Традиційний підхід до розроблення розподілених додатків, що підтримується технологіями розподілених об'єктів, ґрунтується на тісному зв'язку між усіма програмними компонентами. Слабозв’язаність програмних компонентів, яка підтримується технологією веб-сервісів, дає змогу значно спростити координацію розподілених систем та їх реконфігурацію.

6.3. Принципи побудови СОА

Інтероперабельність – здатність двох або більше інформаційних систем (або їх компонентів) до взаємодії з метою вирішення певного завдання й отримання певної інформації.

Це означення об'єднує в собі два поняття:

- *технічна інтероперабельність* – сумісність систем на технічному рівні, включаючи протоколи передання даних і формати їх подання;

- *семантична інтероперабельність* – властивість інформаційних систем, що забезпечує взаємну вживаність отриманої інформації на основі загального розуміння системами її значення.

Прикладом семантичної інтероперабельності програмних систем може бути процес передання певних даних у текстовому вигляді по каналах зв'язку. Наприклад, якщо системи є семантично неінтероперабельними, то одержувач не зможе однозначно інтерпретувати отриманий рядок «1.23»: це може бути число з плаваючою комою, записане в десятковій або шістнадцятковій системі числення, а може бути дата, яку потрібно інтерпретувати як «23 січня».

СОА не задає жорсткої вертикальної (зверху вниз) методології проектування, упровадження ІТ-інфраструктури або керування нею. СОА обмежується кількома принципами, які характеризують кожен з цих процесів; тому її іноді називають не архітектурою, а архітектурним стилем.

Основні принципи побудови СОА:

1. ***Розподілене проектування.*** Рішення щодо внутрішніх особливостей інформаційних систем приймаються різними групами людей, які мають власні організаційні, політичні й економічні мотиви.

2. ***Постійність змін.*** Окремі ділянки архітектури можуть зазнавати змінень у будь-який момент часу.

3. ***Послідовне вдосконалення.*** Локальне поліпшення компонентів архітектури має приводити до вдосконалення всієї архітектури в цілому – до зростання сумарної корисності компонентів того ж рівня, що й змінний, так само, як і компонентів нижчого й вищого рівнів.

Наприклад, відомий веб-сервіс Google Translate постійно зазнає змін. Спочатку він забезпечував тільки веб-інтерфейс для перекладу й обмежений набір мов. Поступово збільшувалися функціональні можливості сервісу: розширювався набір мов, з'явилася можливість голосового відтворення перекладу, при перекладі окремого слова почали видаватися словникові статті з декількома результатами перекладу і т. ін. При цьому API (Application Programming Interface) та інтерфейс змінилися незначно.

4. ***Рекурсивність.*** Однотипні рішення мають місце на різних рівнях архітектури.

6.4. Підхід СОА

З огляду на інформаційні технології логіка підприємства може бути поділена на бізнес-логіку (БЛ) і логіку додатка (ЛД) (рис. 6.1).

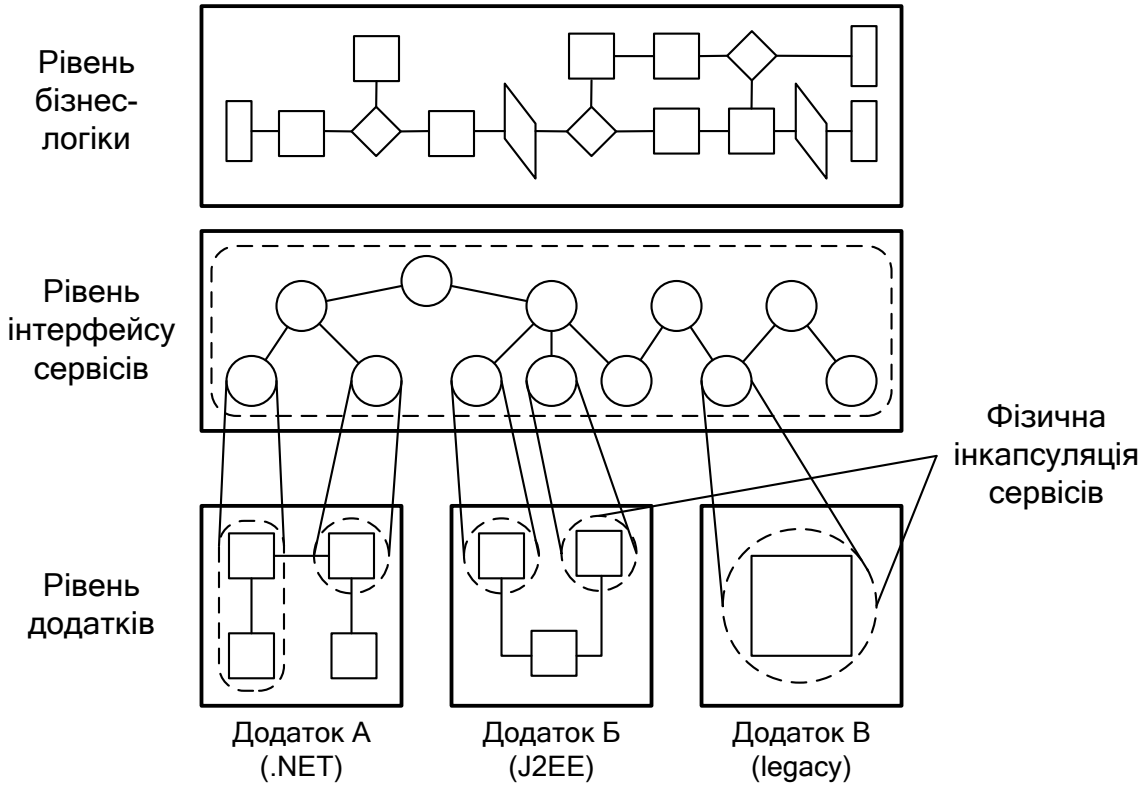

Рис. 6.1. Рівні логіки підприємства

Бізнес-логіка – документальна реалізація бізнес-вимог, що виходять з проблемної області, у якій працює підприємство. БЛ зазвичай є структурованою в процесах, що виражають ці вимоги, а також обмеження й залежності від зовнішніх впливів.

Логіка додатка – це реалізація БЛ, організована на основі різних технологічних рішень. ЛП виражає процеси БЛ з допомогою придбаних або спеціально розроблених програмних систем в умовах обмежених технічних можливостей і залежностей від постачальника рішення.

Перетворення бізнес-логіки на логіку додатків і реалізація сервісів на основі певних вимог є процесами створення сервісно-орієнтованої інфраструктури для вирішення завдань підприємства. Не існує «догматичних» принципів побудови СОА, але при реалізації власної інфраструктури бажано дотримуватися деяких основних принципів.

1. ***Сервіси мають підтримувати повторне використання.*** СОА-системи мають підтримувати повторне використання всіх сервісів незалежно від сьогочасних вимог до їх функціональних особливостей. Якщо під час розроблення системи постаратися максимально врахувати цю вимогу, то підвищуються шанси значно спростити процес вирішення завдань,

що неодмінно виникатимуть у майбутньому при розвитку системи. Сервіс, спочатку орієнтований на повторне використання, дає змогу уникнути розроблення «обгортки», яка б підбудовувала старий сервіс для вирішення нових завдань.

2. ***Сервіси мають забезпечувати формальний контракт використання***, що надає інформацію:

- про кінцеву точку (service endpoint): адресу, за якою можна звернутися до певного сервісу;
- всі операції, що надаються сервісом;
- всі повідомлення, що підтримуються кожною операцією;
- правила й характеристики сервісу та його операцій.

3. ***Сервіси мають бути слабозв'язаними.*** Ніхто не може передбачити, у який бік буде розвиватися IT-інфраструктура. Рішення можуть розвиватися, взаємодіяти, заміняти одне одне. У зв'язку з цим основним завданням є збереження цілісності системи в межах такого розвитку незалежно від змін, що відбуваються.

Система сервісів є слабозв'язаною, якщо сервіс може здобувати знання про інший сервіс, залишаючись незалежним від внутрішньої реалізації логіки цього сервісу. Це досягається шляхом використання контрактів сервісів.

Слабозв'язаність програмних компонентів, що є основою СОА, дає змогу значно спростити координацію розподілених систем та їх реконфігурацію.

4. ***Сервіси мають абстрагувати внутрішню логіку.*** Кожен сервіс має діяти як «чорний ящик», що приховує деталі від навколишнього світу. Немає чіткого визначення, який обсяг логіки має поміщатися в окремому сервісі, проте однією з вимог забезпечення слабкої зв'язаності є взаємодія сервісів на рівні інтерфейсів.

5. ***Сервіси мають бути сумісними.*** Сервіс може реалізовувати логіку як самостійно, так і з застосуванням інших сервісів. Сервіси мають бути спроектовані таким чином, щоб їх можна було використовувати як елементи іншого сервісу (рис. 6.2). Принцип сумісності не залежить від того, чи використовує сервіс для виконання своєї роботи інші сервіси.

Сумісність – це, по суті, просто інша форма повторного використання, і тому операції мають бути стандартними, а для найбільшої сумісності мати необхідний рівень деталізації.

6. ***Сервіси мають бути автономними.*** Властивість автономності потребує, щоб області бізнес-логіки й ресурсів, що використовуються сервісом, мали явні межі, що дає змогу сервісу самому керувати всіма своїми процесами. Це усуває залежність від інших сервісів, що звільняє сервіс від зв'язків, які можуть перешкоджати його застосуванню і розвитку. Питання автономності – найважливіший аргумент під час розподілу бізнес-логіки на окремі сервіси.

Автономність не обов'язково надає сервісу виключне право власності на бізнес-логіку, яку він інкапсулює. Існує два типи автономності:

- *автономність на рівні сервісу:* межі відповідальності сервісів є відокремленими, але вони можуть використовувати загальні ресурси;

- *чиста автономність:* бізнес-логіка й ресурси перебувають під повним контролем сервісу; зазвичай такий вид автономності використовується, коли для реалізації сервісу бізнес-логіка створюється з нуля.

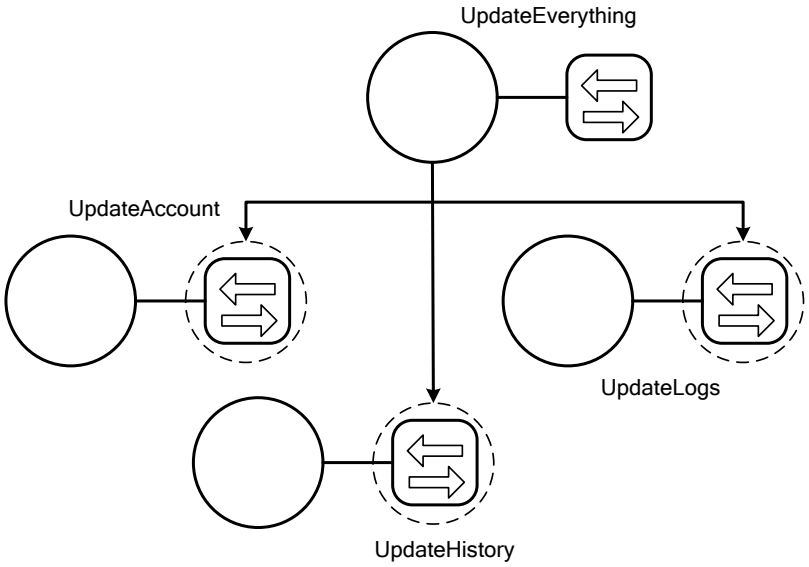

Рис. 6.2. Сервіси, що використовуються як елементи іншого сервісу

7. ***Сервіси не мають використовувати інформацію про стан.*** Сервіси мають зводити до мінімуму обсяг інформації про стан і час, протягом якого вони нею володіють. Інформація про стан – це певні дані, що характеризують поточну діяльність. Наприклад, поки сервіс обробляє повідомлення, він тимчасово залежить від стану (stateful). Якщо сервіс відповідає за збереження стану протягом більш тривалого часу, то його здатність залишатися доступним для інших клієнтів буде утрудненою.

Незалежність від стану (statelessness) дає змогу підвищити можливість масштабованості й повторного використання сервісів. Операції сервісу мають розроблятися з урахуванням міркувань щодо оброблення інформації без даних про стан.

Для підтримки незалежності від стану в СОА використовуються повідомлення-документи. Чим складніше повідомлення, тим більш незалежним і самодостатнім воно залишається.

8. ***Сервіси мають підтримувати виявлення.*** Виявлення сервісів дає змогу уникнути випадкового створення надлишкового сервісу, що забезпечує надлишкову логіку. Метадані сервісу мають детально описувати не тільки загальну мету сервісу, але й функціональність, реалізовану його операціями.

На рівні СОА виявлення характеризує здатність архітектури забезпечити механізми пошуку, такі як реєстр або каталог. На рівні сервісу принцип виявлення належить до процесу проектування окремого сервісу так, щоб цей сервіс настільки піддавався виявленню, наскільки це можливо.

7. ВЕБ-СЕРВІСИ

Веб-сервіси (веб-служби) – це програмні компоненти, з допомогою яких можна створювати незалежні масштабовані слабозв'язані додатки.

Веб-сервіс – це програмна система зі стандартизованими інтерфейсами, що ідентифікується веб-адресою. Веб-служба є одиницею модульності при використанні *сервіс-орієнтованої архітектури* додатку.

Специфікація визначає три основні стандарти, які використовуються для підтримки подання, пошуку й обміну інформацією між веб-сервісами: WSDL, UDDI і SOAP. Ці стандарти утворюють так званий трикутник СОА (рис. 7.1) [5]:

- ***замовник*** (клієнт сервісу, service requestor);
- ***виконавець*** (постачальник сервісу, service provider);
- ***каталог*** (реєстр, service broker).

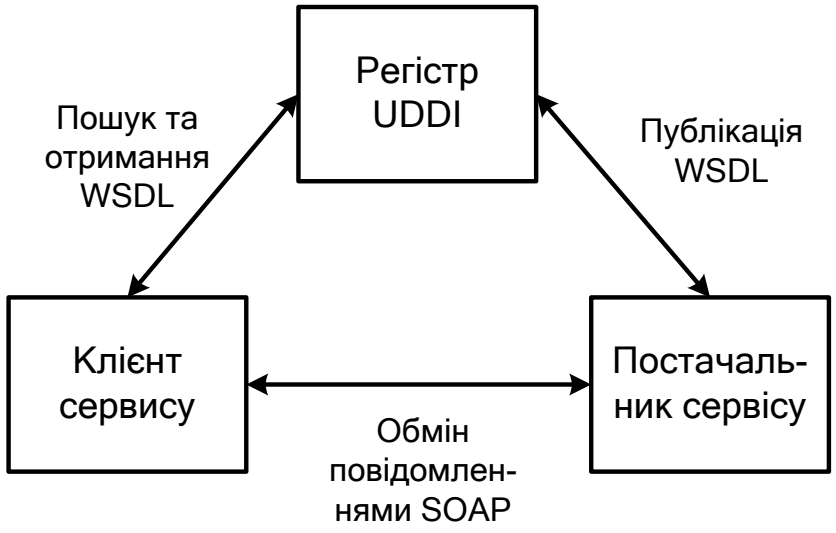

Рис. 7.1. Процес взаємодії між клієнтом і постачальником веб-сервісу (трикутник СОА)

Коли службу розроблено, виконавець реєструє її в каталозі, де її можуть знайти потенційні замовники. Замовник, знайшовши в каталозі відповідну службу, імпортує звідти її WSDL-специфікацію й розробляє відповідно до неї своє програмне забезпечення. WSDL описує формат запитів і відповідей, якими обмінюються замовник і виконавець у процесі роботи.

7.1. Стандарти забезпечення взаємодії веб-сервісів

Для забезпечення взаємодії веб-сервісів використовують такі нормативні документи:

- ***XML*** (EXtensible Markup Language) – розширювана мова розмічання, призначена для зберігання й передання структурованих даних;
- ***SOAP*** – протокол обміну повідомленнями на базі XML;
- ***WSDL*** (Web Services Description Language) – мова опису зовнішніх інтерфейсів веб-служби на базі XML;
- ***UDDI*** (Universal Discovery, Description and Integration) – універсальний інтерфейс розпізнавання, опису та інтеграції; каталог веб-служб і відо-

мостей про компанії, що надають веб-служби в загальне користування або конкретним компаніям [5].

Стек протоколів, з яких складається архітектура веб-сервісів, містить чотири рівні (рис. 7.2):

- процес;
- опис;
- повідомлення;
- зв'язок.

При цьому кожен верхній рівень спирається на нижній рівень.

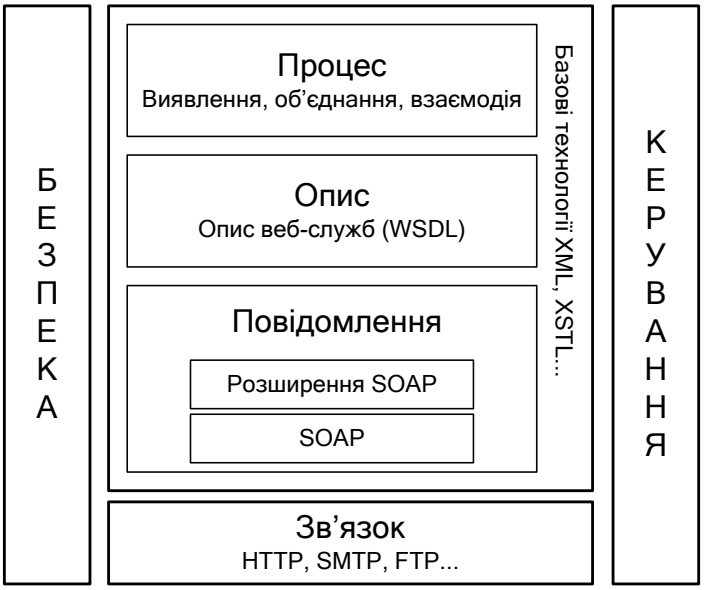

Рис. 7.2. Стек протоколів веб-сервісів

7.1.1. Стандарт XML

Мова розмічання документів – це набір спеціальних інструкцій – тегів, призначених для формування в документах якоїсь структури й визначення зв'язків між різноманітними елементами цієї структури. Теги мови якимось чином кодуються, виділяються відносно основного вмісту документа і є інструкціями для програми, що показує вміст документа на стороні клієнта. У найперших системах для позначення цих команд використовувалися символи <та>, усередині яких містилися назви інструкцій та їх параметри [30].

З 1991 року у всесвітній павутині для розмічання документів використовується мова HTML (HyperText Markup Language), яка набула великої популярності завдяки своїй простоті і зручності використання. Однак через байдужість до структури документа, що унеможливлювало використання тегів для пошуку потрібних фрагментів документа, та обмеженість набору тегів, що не давало можливості швидко адаптувати мову для відображення специфічної інформації, з розвитком технологій HTML ця мова повною мірою не задовольняє вимоги розробників додатків. Тому було запропоновано нову мову гіпертекстового розмічання XML, що має такі переваги:

1) можливість визначення власних команд;
2) можливість використання як універсальної мови запитів до сховищ інформації;
3) можливість здійснення контролю за правильністю даних і перевірки ієрархічних співвідношень усередині документа;
4) невисока складність програм-обробників XML-документів;

5) наявність готових вільно поширюваних програмних продуктів для роботи з XML-документами.

На рис. 7.3 показано приклад найпростішого XML-документа.

Формально правильний XML-документ має відповідати таким вимогам:

- у заголовку документа має бути оголошення XML, у якому вказуються мова розмічання документа, номер її версії і додаткова інформація;
- кожен відкривальний тег, що визначає деяку область даних у документі, обов'язково повинен мати закривального "напарника";
- у XML має враховуватися регістр символів;
- усі значення атрибутів, що використовуються у визначенні тегів, мають подаватися в лапках;
- порядок проходження відкривальних і закривальних тегів має відстежуватися, оскільки вкладеність тегів у XML строго контролюється.

```
<? Xml version = "1.0"?>
<List_of_items>
<Item id = "1"> <first /> Перший </ item>
<Item id = "2"> Другий <sub_item> підпункт 1 </ sub_item> </
item>
<Item id = "3"> Третій </ item>
<Item id = "4"> <last /> Останній </ item>
</ List_of_items>
```

Рис. 7.3. Приклад найпростішого XML-документа

Уміст XML-документа являє собою набір елементів, секцій CDATA, директив аналізатора, коментарів, спецсимволів, текстових даних.

Елемент – це структурна одиниця XML-документа. Укладаючи слово *rose* в теги <flower> </ flower>, визначаємо непорожній елемент під назвою <flower>, умістом якого є *rose*. У загальному випадку вмістом елементів можуть бути практично будь-які частини XML-документа.

Коментарями є будь-яка область даних, поміщена між послідовностями символів <! - і ->. Коментарі пропускаються аналізатором, і тому при розборі структури документа як значуща інформація не розглядається.

Атрибут – це пара назва = "значення", яка задається при визначенні елемента в початковому тегу. Атрибути використовуються у випадку, коли необхідно задати параметри, що уточнюють характеристики елемента. Приклад задання атрибута: <author id = 0> Ivan Petrov </ author>.

Безкоштовний домен. Для того щоб включити в документ символ, що використовується для визначення будь-яких конструкцій мови (наприклад, символ кутової дужки), і не спричинити при цьому помилок у процесі розбору такого документа, потрібно використовувати його спеціальний символьний або числовий ідентифікатор (наприклад, & lt; & gt ; & quot).

Директиви аналізатора – це інструкції, що описуються з допомогою спеціальних тегів - <? і ?> і призначені для аналізаторів мови. Програма клієнта використовує ці інструкції для керування процесом розбору документа. Найбільш часто інструкції застосовуються при визначенні типу документа (наприклад, <? Xml version = "1.0 "?>) або створенні простору імен [30].

CDATA. Щоб задати область документа, яку при розборі аналізатор буде розглядати як простий текст, ігноруючи будь-які інструкції та спеціальні символи, але на відміну від коментарів мати можливість використовувати їх в додатку, необхідно використовувати теги <! [CDATA] i]]>. Усередині цього блока можна поміщати будь-яку інформацію, що може знадобитися програмі-клієнту для виконання будь-яких дій (в область CDATA можна поміщати, наприклад, інструкції JavaScript). Природно, потрібно стежити за тим, щоб в області, обмеженій цими тегами, не було послідовності символів]].

7.1.2. Стандарт SOAP

Стандарт SOAP призначено для організації взаємодії віддалених систем з допомогою асинхронного обміну повідомленнями [5].

Повідомлення SOAP забезпечують однобічне передання інформації (від джерела до приймача) між вузлами SOAP. Ці повідомлення є основним блоком, що забезпечує можливість побудови більш складних шаблонів взаємодії: запит/відповідь, "діалоговий" режим тощо.

Повідомлення SOAP є XML-відформатованим документом, який складається з конверта, що містить заголовок, і тіла повідомлення (рис. 7.4).

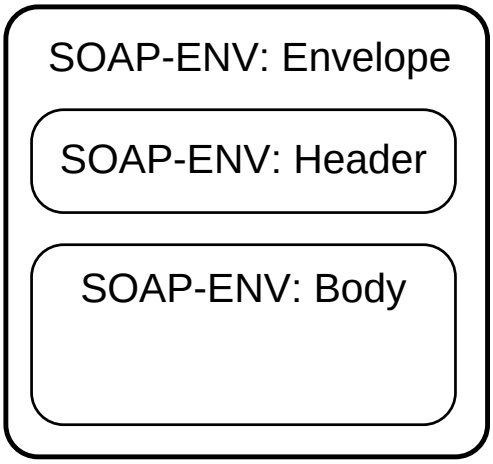

Рис. 7.4. Структура SOAP-повідомлення

У ***тілі*** міститься XML-блок з інформацією, яка має бути доставлена кінцевому адресату.

Заголовок – необов'язковий елемент, з допомогою якого можна передавати дані, які не є основним робочим навантаженням (наприклад, інформацію для оброблення повідомлення). SOAP-повідомлення може йти за маршрутом, який містіть кілька вузлів, кожен з яких може якось його обробляти. Статус цих змін відображується в блоках заголовка повідомлення. Приклад заголовка SOAP-повідомлення зображено на рис. 7.5.

```
<Soap: Header>
        <Trans: Transaction
        xmlns: trans =
"http://www.host.com/namespaces/space/"
        soap: mustUnderstand = "1">
        12
        </ Trans: Transaction>
</ Soap: Header>
```

Рис. 7.5. Приклад заголовка SOAP-повідомлення

У заголовку SOAP-повідомлення можна ввести нові елементи, що не передбачені стандартом SOAP. Ці елементи відіграють утилітарну роль щодо основного повідомлення, що міститься в тілі SOAP (наприклад, номер транзакції, протягом якої надійшло повідомлення; інформація для авторизації користувача тощо).

Для елементів заголовка можна вказати значення атрибутів:

► *mustUnderstand* – якщо значення дорівнює одиниці, то одержувач повинен обробити цей елемент заголовка; якщо одержувач не вміє цього робити, то він має відкинути це повідомлення;

► *actor* – указує назву конкретного додатка-одержувача, якщо SOAP-повідомлення проходить ланцюжок додатків під час оброблення.

На рис. 7.6 наведено приклад реалізації запиту й відповіді з допомогою SOAP-повідомлень.

Запит

```
<Soap: Envelope xmlns: soap = "http://schemas.xmlsoap.org/soap/envelope/"> <Soap:
Body>
        <GetProductDetails xmlns = "http://warehouse.example.com/ws">
        <ProductID> 12345 </ productID>
        </ GetProductDetails>
    </ Soap: Body>
 </ Soap: Envelope>
```

Відповідь

```
<Soap: Envelope xmlns: soap = "http://schemas.xmlsoap.org/soap/envelope/">
    <Soap: Body>
        <getProductDetailsResponse
        xmlns = "http://warehouse.example.com/ws">
            <GetProductDetailsResult>
                <ProductID> 12345 </ productID>
                <Produc tName> Router NoName </ produc tName>
                <Description> Router NoName, WiFi, LAN </ description>
                  <Price> 55 0 </ price>
                <InStock> true </ inStock>
            </ GetProductDetailsResult>
        </ GetProductDetailsResponse>
    </ Soap: Body>
     </ Soap: Envelope>
```

Рис. 7.6. Приклад запиту і відповіді з допомогою SOAP-повідомлень

У тілі SOAP-повідомлення відбувається передання повідомлення за форматом, визначеним у блоках <portType> і <message> WSDL-документа. Ім'я основного блока, що знаходиться в тілі SOAP-повідомлення, відповідає імені повідомлення, яке визначили в інтерфейсі веб-сервісу.

Стандарт SOAP забезпечує однобічне передання повідомлень, тому для отримання відповіді від сервера, якому було передано SOAP-повідомлення, можуть знадобитися ініціалізація процесу передання повідомлення і встановлення з'єднання з клієнтом. Це може викликати значні труднощі в сучасних умовах організації зв'язку в мережі інтернет, оскільки більшість клієнтів не мають виділених статичних IP-адрес.

Цю проблему вирішено в межах стандартного зв'язування протоколу SOAP і протоколу HTTP (англ. *SOAP HTTP Binding*), що реалізує патерн поведінки «запит – відповідь». Приклад реалізації патерна «запит – відповідь» з допомогою HTTP-зв'язування показано на рис. 7.7.

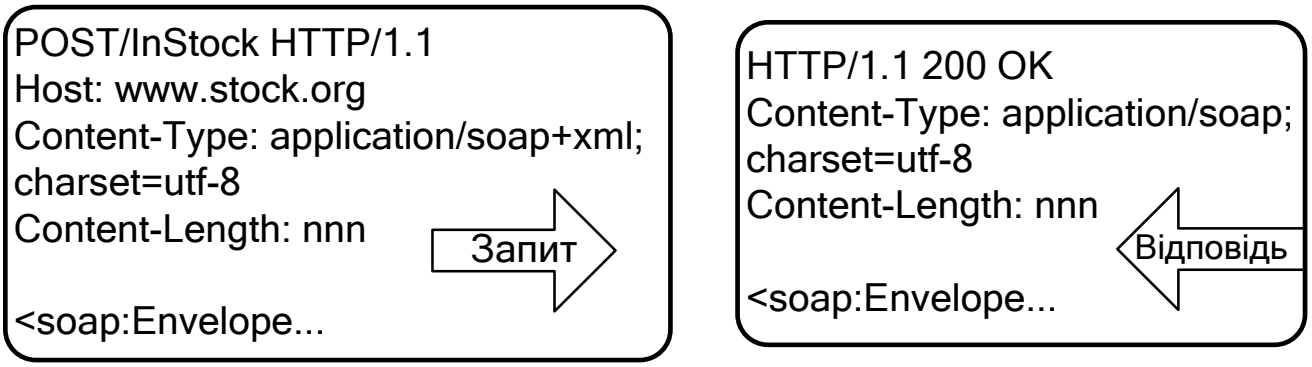

Рис. 7.7. Приклад реалізації патерна «запит – відповідь» з допомогою HTTP-зв'язування

У заголовку «Content-Type» для повідомлень HTTP-запиту і HTTP-відповіді встановлюється значення text/xml (application/soap+xml в SOAP 1.2). HTTP-запит повинен використовувати POST (починаючи з SOAP 1.2, можна використовувати GET). HTTP-відповідь має використовувати статусний код 200, якщо оброблення SOAP-повідомлення пройшло нормально, або 500, якщо в тексті повідомлення міститься помилка SOAP.

7.1.3. Стандарт WSDL

Стандарт WSDL описує сервіси у вигляді якихось абстрактних ресурсів, здатних приймати на вхід документи певних типів та ініціювати відправлення документів інших типів. WSDL використовується для опису веб-сервісів і визначення їх розташування. WSDL написано мовою XML і він є XML-документом.

WSDL визначає сервіс з двох точок зору:

- *абстрактної:* сервіс задається в термінах повідомлень, що посилаються і приймаються ним, та описуються засобами XML Schema у вигляді, що не залежить від конкретного транспортного протоколу;

- *конкретної:* визначаються прив'язки до транспортних форматів і точок фізичного розміщення.

Група специфікацій WSDL 2.0 складається з трьох основних документів:

- WSDL Part 1: Core language («Основна мова»);
- WSDL Part 2: Message exchange patterns («Шаблони обміну повідомленнями»);
- WSDL Part 2: Bindings («Прив'язки»).

```
public class MyMath
{
        public int squared (int x)
        {
                return x * x;
        }
}
```

Рис. 7.8. Приклад вихідного коду для тестового веб-сервісу

Розглянемо приклад найпростішого веб-сервісу, який буде забезпечувати піднесення до квадрата переданого числа. Мовою Java цей сервіс можна було б описати у вигляді вихідного коду, показаного на рис. 7.8.

Припустимо, що даний веб-сервіс будемо розміщувати в інтернеті за адресою http://supercomputer.susu.ru/MyMath. Для того щоб будь-який клієнт міг отримати інформацію про те, які методи надає веб-сервіс і як до них необхідно звертатися, використовується WSDL-документ, що описує повну специфікацію методів взаємодії з зазначеним сервісом. Отже, необхідно сформувати (вручну або автоматично) WSDL-документ і помістити його за адресою http://supercomputer.susu.ru/MyMath.wsdl.

Стандарт WSDL забезпечує опис веб-сервісу у вигляді повідомлень, які може відправити або ж прийняти веб-сервіс, а також відповідає за методи зв'язування даних повідомлень з базовим середовищем передання даних. У зв'язку з цим виділяють такі елементи WSDL-документа:

- *блок types* – типи даних, що використовуються веб-сервісом;
- *блок message* – повідомлення, що використовуються веб-сервісом;
- *блок portType* – методи, які звичайно пропонуються сервісом;
- *блок binding* – протоколи зв'язку, які використовуються веб-сервісом.

Елемент <portType> – найважливіший елемент WSDL, що визначає сам веб-сервіс, операції, що ним надаються, і використовувані повідомлення. Цей елемент можна порівняти з бібліотекою функцій, у якій указано вхідні параметри й результати роботи функції.

Розглянемо, яким чином буде описуватися сервіс MyMath (рис. 7.9). Інтерфейс сервісу MyMath складається з однієї операції squared, яка повинна виконуватися з одним параметром in0. Ця операція складається з двох повідомлень:

- *вхідного* (Wsdl: input message = "impl: squaredRequest"), яке має передати користувач веб-сервісу в цей сервіс для того, щоб запустити операцію піднесення до квадрата;

- *вихідного* (Wsdl: output message = "impl: squaredResponse"), яке буде повернуто користувачеві після того, як операція піднесення до квадрата успішно завершиться.

```
<Wsdl: portType name = "MyMath">
        <Wsdl: operation name = "squared" parameterOrder = "in0">
                <Wsdl: input message = "impl: squaredRequest"
         name = "squaredRequest" />
                <Wsdl: output message = "impl: squaredResponse"
 name = "squaredResponse" />
        </ Wsdl: operation>
</ Wsdl: portType>
```

Рис. 7.9. Приклад опису блока portType

Елемент <message> визначає елементи даних операції (рис. 7.10). Кожне повідомлення може містити одну або кілька частин. Ці частини можна порівняти з параметрами виклику функцій у традиційних мовах програмування.

```
<Wsdl: message name = "squaredRequest">
        <Wsdl: part name = "in0" type = "xsd: int" />
</ Wsdl: message>
<Wsdl: message name = "squaredResponse">
        <Wsdl: part name = "squaredReturn" type = "xsd: int" />
</ Wsdl: message>
```

Рис. 7.10. Приклад опису блока message

У <message> наводиться опис усіх частин повідомлень "squared-Request" і "squared-Response", інтерфейс яких описано в блоці <portType>. Кожна частина повідомлення – це параметр виклику методу сервісу. Для проведення операції MyMath користувач має передати один вхідний параметр "in0" у вигляді цілого числа, що визначається атрибутом type = "xsd: int". Результатом операції також буде ціле число відповідно до змісту блока <wsdl: part name = "squaredReturn" type = "xsd: int" />.

Елемент <binding> визначає формат повідомлення й деталі протоколу для кожного порту (рис. 7.11) і відповідає за те, яким чином елементи абстрактного інтерфейсу в блоці <portType> перетворюються на масиви інформації в форматі протоколів взаємодії, наприклад SOAP.

Основною частиною блока <binding> є елемент <soap: binding>, що визначає конкретний протокол передання даних. Атрибут style визначає тип запиту і може мати два значення: rpc (Remote Procedure Call – віддалений виклик процедур) і document. У блоці <soap: binding> також оголошується transport – визначальний протокол, на основі якого буде проводитися взаємодія (зазвичай HTTP).

```
<Wsdl: binding name = "MyMathSoapBinding" type = "impl: MyMath">
        <Wsdlsoap: binding style = "rpc"
        transport = "http://schemas.xmlsoap.org/soap/http"/>
        <Wsdl: operation name = "squared">
                <Wsdlsoap: operation soapAction = "" />
                <Wsdl: input name = "squaredRequest">
                        <Wsdlsoap: body encodingStyle =
                        "http://schemas.xmlsoap.org/soap/encoding/"
                        namespace = "http: // DefaultNamespace"Use =" encoded
                "/>
                </ Wsdl: input>
                <Wsdl: output name = "squaredResponse">
                         <Wsdlsoap: body encodingStyle =
                        "http://schemas.xmlsoap.org/soap/encoding/"
                        namespace = "http: // DefaultNamespace"Use =" encoded
                "/>
                </ Wsdl: output>
        </ Wsdl: operation>
</ Wsdl: binding>
```

Рис. 7.11. Приклад опису блока binding

Усередині <soap: operation> міститься елемент, який описує значення поля soapAction HTTP-запиту.

Елементи input і output визначають, як будуть декодуватися вхідні і вихідні повідомлення цієї операції.

У блоках <port> і <service> відбувається визначення, де знаходиться сервіс:

- *port* описує розташування і спосіб доступу до кінцевої точки;
- *service* – іменована колекція портів.

Як видно з прикладу (рис. 7.12), у блоці <port> не відбувається безпосереднього опису методів взаємодії з веб-сервісом. Вони описуються раніше, у блоці <binding>, а в блоці <port> тільки дається посилання на описаний метод зв'язку binding.

```
<Wsdl: service name = "MyMathService">
        <Wsdl: port binding = "impl: MyMathSoapBinding" name = "MyMath">
                <Wsdlsoap: address location = "http://supercomputer.susu.ru/MyMath"/>
        </ Wsdl: port>
 </ Wsdl: service>
</ Wsdl: definitions>
```

Рис. 7.12. Приклад блока service

7.1.4. Стандарт UDDI

UDDI являє собою стандарт на внутрішню будову і зовнішні інтерфейси бази даних (сховища), що зберігає опис сервісів. Усі описи в БД зберігаються у вигляді XML-записів.

UDDI є відкритим проектом, що спонсується OASIS (Organization for the Advancement of Structured Information Standards) і дає змогу організаціям публікувати WSDL-описи веб-сервісів для подальшого їх пошуку іншими організаціями й інтеграції у власні системи [31].

Реєстрація UDDI складається з таких трьох компонентів:

- ***білі сторінки***, що надають інформацію про постачальника послуг, наприклад, назва компанії, опис послуги (можливо, декількома мовами), використовуючи яку, можна знайти службу, частина відомостей про яку є вже відомою (наприклад, розміщення сервісу, яке знайшли за ім'ям провайдера);
- ***жовті сторінки***, що містять класифікацію служби або бізнесу на основі стандартних (Standard Industrial Classification (SIC), North American Industry Classification System (NAICS), United Nations Standard Products and Services Code (UNSPSC)) і географічних таксономій; якщо бізнес надає ряд послуг, то може бути кілька жовтих сторінок (кожна з яких описує послугу), пов'язаних з однією білою сторінкою;
- ***зелені сторінки***, що використовуються для опису способу отримання доступу до веб-служб та інформації про зв'язані послуги; частина інформації, що публікується на зелених сторінках, пов'язана з веб-сервісами, наприклад, адреса сервера, параметри й посилання на специфікації інтерфейсів, інша частина інформації, безпосередньо не пов'язана з веб-службою, вона містить електронну пошту, телефонні номери та іншу інформацію для певного сервісу; якщо в WSDL-описі сервісу визначено кілька прив'язок, то цей сервіс може мати кілька зелених сторінок.

7.2. Друге покоління стандартів веб-сервісів

Після появи технології веб-сервісів на початку 2000-х років багато розробників відчули, що відсутність стандартизації в найбільш важливих областях побудови РВС призводить до несумісності розроблюваних рі-

шень. Для вирішення цих проблем безліч комерційних і некомерційних організацій об'єднали свої зусилля в межах різних консорціумів для розроблення нового покоління стандартів веб-сервісів (WS-стандартів). Найбільш визнаними й поширеними серед WS-стандартів є:

- **WS-Security**, пов'язаний із забезпеченням безпеки веб-сервісів;
- **WS-Addressing**, який розглядає питання маршрутизації і адресації SOAP-повідомлень;
- **WSRF, WS-Notification**, що описують роботу зі станом веб-сервісів.

7.2.1. WS-Security

WS-Security є базою для інших технологій в області безпеки веб-сервісів.

Стандарт WS-Security є орієнтованим на комплексне вирішення завдань безпеки при взаємодії веб-сервісів і забезпечує визначення основних методів ідентифікації користувача, цифрові підписи, шифрування.

WS-Security забезпечує переміщення завдань ідентифікації та авторизації в область обміну SOAP-повідомленнями. Тепер інформація, що міститься в SOAP-повідомленні, має забезпечувати:

- ідентифікацію категорій користувачів, пов'язаних з повідомленням;
- доведення того, що категорії користувачів мають правильний набір прав доступу;
- доведення того, що повідомлення не змінювалося.

WS-Security дає можливість застосовувати маркери безпеки (МБ) (Security Tokens) при роботі з SOAP-повідомленнями (рис. 7.13).

Використовуючи МБ, SOAP-повідомлення може містити таку інформацію:

- *функція ідентифікації:* я – User Vasya Pupkin;

Рис. 7.13. Процедура авторизації користувача на основі WS-Security

- *належність до групи:* я – розробник PupkinSite.com;
- *підтвердження прав:* оскільки я – розробник PupkinSite.com, то я можу створювати бази даних і додавати веб-додатки в сервери PupkinSite.com.

Стандарт WS-Security передбачає безліч різних способів перевірки достовірності користувача, основні серед яких:

- <Wsse: UsernameToken> – аутентифікація користувача з допомогою пари «Ім'я користувача/пароль»;
- <Wsse: X509v3> – аутентифікація з допомогою сертифіката X.509v3;
- *Kerberos* – аутентифікація з допомогою протоколу Kerberos (Kerberos Domain Controller) (використовується в Windows2000, Red Hat Linux і т. ін.).

Ім'я користувача/пароль. Для передання посвідчення користувача таким способом у WS-Security визначено елемент usernameToken, який використовує два інших типи: username і Password. Способи передання пароля:

- *як простий текст* – зазвичай застосовується, якщо обмін SOAP-повідомленнями ведеться поверх установленого захищеного з'єднання;
- *у цифровому форматі* – передбачає хешування на основі випадкового ключа та інформації про час формування повідомлення; такий алгоритм можна використовувати навіть при обміні повідомленнями по відкритому каналу.

Аутентифікація на основі сертифіката X.509. Коли повідомлення посилає сертифікат X.509, воно передає відкриту версію сертифіката в маркер WS-Security BinarySecurityToken. Сам сертифікат отримує відправлене повідомлення як шифровані base64 дані. Для забезпечення безпеки при використанні сертифіката потрібно вжити додаткових засобів забезпечення безпеки: підпис повідомлення секретним ключем сертифіката й додавання wsu:Timestamp для визначення часу життя повідомлення.

Kerberos. Модель містить два логічних компоненти: сервер аутентифікації (CA) і сервер видачі квитків (TGS – Ticket Granting Server), які зазвичай постачаються як єдина програма, що запускається на центрі розподілу ключів (ЦРК містить базу даних логінів/паролів для користувачів і сервісів, що використовують Kerberos).

1. CA отримує запит AS_REQ, що містить ім'я клієнта, який запитує аутентифікацію, і повертає йому зашифрований TGT (Ticket Granting Ticket – квиток на отримання квитка), який може використовуватися для запиту подальших квитків на інші сервіси. У більшості реалізацій Kerberos час життя TGT становить 8–10 годин.
2. CA перевіряє AS_REQ і генерує випадковий сеансовий ключ, який буде спільно використовуватися клієнтом і TGS. ЦРК створює дві копії сесійного ключа: для клієнта і TGS. Ключ відправляється клієнту у вигляді повідомлення, зашифрованого довгостроковим ключем клієнта.

3. Якщо користувач захоче отримати доступ до сервісу, то він підготує повідомлення для TGS (TGS_REQ), що містить три частини: ідентифікатор сервісу, копію TGT, отриману раніше, і аутентифікатор.

4. При отриманні запиту квитка від клієнта ЦРК формує новий сесійний ключ для взаємодії клієнт/сервіс і в зашифрованому вигляді відправляє повідомлення (TGS_REP) клієнту.

Підпис повідомлення дає змогу одержувачеві SOAP-повідомлення впевнитися в тому, що підписані елементи не були змінені під час передання повідомлення. Однак підпис сам по собі не може захистити повідомлення від перегляду його вмісту третіми особами.

За підпис повідомлення відповідає специфікація XML Signature. Залежно від вибраного методу аутентифікації в процесі підписування повідомлення може бути використана така інформація:

- *UsernameToken* ⇨ пароль користувача;
- *X.509* ⇨ секретний ключ;
- *Kerberos* ⇨ сеансовий ключ.

За шифрування відповідає стандарт XML Encryption. Згідно з цим стандартом увесь уміст тіла SOAP-повідомлення шифрується і замінюється на блок <xenc:EncryptedData> (рис. 7.14), який може бути прочитаний тільки за наявності секретної авторизаційної інформації.

```
<soap: Envelope>
        ...
<soap: Body>
       <Xenc: EncryptedData
        Id = "EncryptedContent-f6f50b24-3458-41d3-aac4-390f476f2e51"
        Type = "http://www.w3.org/2 0 01/04 / xmlenc # Content ">
        <Xenc: EncryptionMethod Algorithm =
        "http://www.w3.org/2 0 01/04 / xmlenc # tripledes-cbc "/>
        <KeyInfo xmlns = "http://www.w3.org/2 00 0/0 9 / xmldsig # ">
 <KeyName> Symmetric Key </ KeyName>
 </ KeyInfo>
 <Xenc: CipherData>
        <Xenc: CipherValue>
        InmSSXQcBV5UiT ... Y7RVZQqnPpZYMg ==
        </ Xenc: CipherValue>
        </ Xenc: CipherData>
       </ Xenc: EncryptedData>
       ...
</ Soap: Envelope>
```

Рис. 7.14. Шифрування повідомлення на основі XML Encryption

7.2.2. WS-Addressing

У стандартах першого покоління повна адреса веб-сервісу містилася в WSDL-описі, у блоці <port>. Це спричиняло значні незручності, тому що

при зміні адреси сервісу доводилося редагувати WSDL-файл цілком. Тому в стандарті WS-Addressing було передбачено введення таких полів в заголовок SOAP-повідомлення:

- ***<Wsa: To>***, що визначає URI приймача повідомлення;
- ***<Wsa: Action>***, що визначає відповідну дію.

Інша незручність, з якою стикалися розробники, полягала в тому, що при обміні SOAP-повідомленнями адресація покладалася на транспортний протокол (при зв'язуванні з HTTP) і не могла бути змінена безпосередньо в SOAP-повідомленні. У зв'язку з цим у WS-Addressing було введено такі поля:

- ***<MessageID>,*** що містить ідентифікатор повідомлення;
- ***<From>***, у якому розміщується адреса відправника;
- ***<ReplyTo>***, що містить адресу отримання відповіді;
- ***<FaultTo>***, у якому вказується адреса реєстрації помилок;
- ***<RelatedTo>***, що містить інформацію для вибудовування повідомлень у ланцюжки.

Крім того, уведено елемент <EndpointReference>, що являє собою розширений варіант блока <Service> WSDL і містить поля:

- <*Wsa:Address/*> – URI, що ідентифікує кінцевий пункт;
- <*Wsa:ReferenceProperties/>,* яке може містити окремі властивості, необхідні для ідентифікації особи або ресурсу (наприклад, ім'я файлу, з яким має вироблятися певна дія, або ідентифікатор кошика покупця);
- <*Wsa:ServiceName PortName = ""/*> – аналог блока ServiceName WSDL;
- <*Wsa:PortType/*> – аналог блока PortType WSDL;
- <*Wsa:Policy/*> – необов'язкове поле, яке може використовуватися для оголошення й реклами політики веб-сервісу в області забезпечення безпеки, якості обслуговування і т. ін.

Завдяки описаним змінам стандарт WS-Addressing забезпечує незалежність методів адресації веб-сервісів і можливість передання і зберігання додаткової інформації в заголовках SOAP-повідомлень.

7.2.3. Стан веб-сервісів і WSRF

Спочатку використання веб-сервісів не передбачало існування «стану». Типовий сценарій використання веб-сервісу базувався на шаблоні «запит – відповідь – відключення». При цьому кожен наступний запит не залежав від результатів попереднього запиту. Однак для багатьох додатків, як комерційних, так і наукових, збереження інформації про стан було істотною вимогою успішного функціонування. Наприклад, для розроблення грід-систем не змогли застосувати «чисті» веб-сервіси, тому що вони не мали можливості працювати зі станом. Відсутність стандартних технологій оброблення стану протягом досить тривалого часу призвело до появи

безлічі несумісних варіантів «симуляції» роботи зі станом з допомогою веб-сервісів.

Специфікація Web Services Resource Framework (WFRF) є спробою вирішити зазначену архітектурну проблему шляхом введення поняття «стан» у веб-сервіси, перетворивши їх на веб-ресурси і вказавши механізми використання цього поняття.

WSRF (Web Service Resource Framework) – це специфікації, які визначають стандартні способи запиту значень властивостей або способи указування того, що ці властивості мають бути змінені.

WSRF уводить поняття WS-ресурсу, що являє собою об'єднання веб-сервісу й ресурсу зі станом, на який веб-сервіс може впливати. WSRF також визначає стандартні способи вирішення багатьох проблемних питань роботи з WS-ресурсами.

Специфікація WSRF містить такі нормативні документи:

- *WS-Resource specification* – опис WS-ресурсів;
- *WS-ResourceProperties (WSRF-RP)* – опис властивостей WS-ресурсів;
- *WS-ResourceLifetime (WSRF-RL)* - опис керування часом життя (створення і знищення) WS-ресурсів;
- *WS-ServiceGroup (WSRF-SG)* – робота з групами ресурсів; визначає спосіб створення набору веб-сервісів (наприклад, регістр наявних сервісів);
- *WS-BaseFaults (WSRF-BF)* – опис основних помилок, які можуть виникнути при роботі з WS-ресурсами.

Розглянемо більш докладно процес створення WS-ресурсів. Стан об'єкта можна визначити через значення його різних властивостей. Ресурс зі станом можна подати у вигляді XML-документа Resource properties document, що містить його властивості.

```
<SatProp: GenericSatelliteProperties
        xmlns: satProp = "http://example.com/satellite">
        <SatProp: latitude> 3 0.3 </ satProp: latitude>
        <SatProp: longitude> 223.2 </ satProp: longtitude>
        <SatProp: altitude> 4 77 0 0 </ satProp: altitude>
        <SatProp: pitch> 4 9 </ satProp: pitch>
        <SatProp: yaw> 0 </ satProp: yaw>
        <SatProp: roll> 32 </ satProp: roll>
        <SatProp: focalLength> 21999992 </ satProp: fo-
calLength>
        <SatProp: currentView>
         http://example.com/satellite/223 9992333.zip
        </ SatProp: currentView>
</ SatProp: GenericSatelliteProperties>
```

Рис. 7.15. Властивості WS-ресурсу «Космічний супутник»

Як приклад розглянемо ресурс «Космічний супутник». Властивості ресурсу (рис. 7.15): широта, довгота, кути нахилу, висота над земною поверхнею і поточний вигляд з об'єктива фотокамери, установленої на цьому супутнику. Це модель ресурсу зі станом. Щоб завершити створення WS-

ресурсу, потрібно зв'язати його з сервісом, використовуючи WSDL-файл. Для цього сформуємо типи даних, які будуть представляти WS-ресурс у WSDL-файлі (рис. 7.16).

```
<Definitions name = "Satellite" ...>
    ...
    <Types>
        <Xsd: schema targetNamespace = "http://example.com/satellite"
        xmlns: xsd = "http://www.w3.org/2001 / XMLSchema ">
            <Xsd: element name = "latitude" type = "xsd: float" />
            <Xsd: element name = "longitude" type = "xsd: float" />
            <Xsd: element name = "altitude" type = "xsd: float" />
            <Xsd: element name = "GenericSatelliteProperties">
            <Xsd: complexType>
                <Xsd: sequence>
                <Xsd: element ref = "latitude" minOccurs = "1" maxOccurs = "1" />
                <Xsd: element ref = "longitude" minOccurs = "1" maxOccurs = "1" />
                </ Xsd: sequence> </ xsd: complexType>
            </ Xsd: element>
      </ Xsd: schema>
     </ Types>
          <PortType name = "SatellitePortType"
              wsrp: ResourceProperties = "tns: GenericSatelliteProperties">
          </ PortType>
</ Definitions>
```

Рис. 7.16. Властивості WS-ресурсу в WSDL-файлі

Як видно з рис. 7.16, спочатку були додані базові елементи веб-сервісу: service і binding, який асоціює service з елементом portType. Сам елемент portType поки не містить жодних операцій, головною його складовою є атрибут wsrp:ResourceProperties. Цей атрибут свідчить про те, що будь-яка операція, яку виконує веб-сервіс, відбувається над певним типом ресурсу зі станом, згідно з тим, як це визначено елементом GenericSatelliteProperties. Сам елемент GenericSatelliteProperties визначено в елементі schema. Об'єднання цього ресурсу зі станом веб-сервісу і є необхідним WS-ресурсом.

Тепер додамо кілька операцій для роботи з ресурсом у базові елементи WSDL-опису веб-сервісу (рис. 7.17). Звернемо увагу на те, що сервіс буде повертати не просте значення, а EndpointReference, де буде міститися посилання на новостворений WS-ресурс, і подивимося, як це буде використано в SOAP-повідомленні.

Фактичного об'єкта поки що немає, тому сам запит направляємо згідно з URI, записаним у WSDL-файлі, а зміст цього запиту визначаємо як простий елемент createSatellite. Після отримання запиту на створення нового супутника сервер створює посилання на новий WS-ресурс і відсилає його назад у формі EndpointReference (рис. 7.18). В елементі

EndpointReference вказано ідентифікатор, який обов'язково повинен використовуватися при ідентифікації WS-ресурсу. Слід зазначити, що значення SatelliteId, як і будь-яка інша інформація, що міститься в посиланні на крайню точку, ніяк не тлумачиться і не обробляється, а просто передається в повідомленнях як «чорний ящик», незалежне життя якого є недоступним для спостережень.

```
<Types>
 <Xsd: element name = "createSatellite"> <xsd: complexType /> </ xsd: element>
 <Xsd: element name = "createSatelliteResponse">
 <Xsd: complexType>
 <Xsd: sequence>
 <Xsd: element ref = "wsa: EndpointReference" />
 </ Xsd: sequence>
 </ Xsd: complexType>
 </ Xsd: element>
</ Types>

<Message name = "CreateSatelliteRequest">
 <Part name = "request" element = "tns: createSatellite">
</ Message>
<Message name = "CreateSatelliteResponse">
 <Part name = "response" element = "tns: createSatelliteResponse" />
</ Message>

<PortType name = "SatellitePortType"
 wsrp: ResourceProperties = "tns: GenericSatelliteProperties">
 <Operation name = "createSatellite">
 <Input message = "tns: CreateSatelliteRequest"
 wsa: Action = "http://example.com/CreateSatellite"/>
 <Output message = "tns: CreateSatelliteResponse"
 wsa: Action = "http://example.com/CreateSatelliteResponse"/>
 </ Operation>
</ PortType>
```

Рис. 7.17. Опис операцій при роботі з WS-ресурсом

Розглянемо приклад запиту на отримання інформації про стан ресурсу (рис. 7.19). Запит містить елемент wsa:Action, який не є частиною створеного посилання на кінцеву точку і змінюється залежно від того, яка дія виконується. У цьому випадку використовуємо дію GetResourceProperty. Елемент wsa:To вибирає значення за адресою wsa:Address (із посилання на крайню точку), і будь-які значення wsa:ReferenceProperty безпосередньо переносяться в заголовок (Header).

Запит:

```
<SOAP-ENV: Envelope xmlns: SOAP-ENV = "http://schemas.xmlsoap.org/soap/envelope/">
 <SOAP-ENV: Header />
 <SOAP-ENV: Body>
        <CreateSatellite xmlns = "http://example.com/satellite"/>
 </ SOAP-ENV: Body>
</ SOAP-ENV: Envelope>
```

Відповідь:

```
<SOAP-ENV: Envelope xmlns: SOAP-ENV = "http: //schemas.xmlsoap. org / soap / envelope / ">
 <SOAP-ENV: Header />
 <SOAP-ENV: Body>
        <Wsa: EndpointReference
        xmlns: wsa = "http://www.w3.org/2 005/02 / addressing "
        xmlns: sat = "http://example.org/satelliteSystem">
                <Wsa: Address>http://example.com/satellite</wsa:Address>
                <Wsa: ReferenceProperties>
                        <Sat: SatelliteId> SAT9928 </ sat: SatelliteId>
                </ Wsa: ReferenceProperties>
        </ Wsa: EndpointReference>
 </ SOAP-ENV: Body>
</ SOAP-ENV: Envelope>
```

Рис. 7.18. Запит на створення WS-ресурсу

```
<SOAP-ENV: Envelope>
      <SOAP-ENV: Header>
            <Wsa: Action>
             http://docs.oasis-open.org/wsrf/2004/06/WS-
             ResourceProperties / GetResourceProperty
            </ Wsa: Action>
            <Wsa: To SOAP-ENV: mustUnderstand = "1">
            http://example.com/satellite </ Wsa: To>
            <Sat: SatelliteId> SAT9928 </ sat: SatelliteId>
      </ SOAP-ENV: Header>
      <SOAP-ENV: Body>
            <Wsrp: GetResourceProperty
             xmlns: satProp = "http://example.com/satellite"> SatProp: altitude
            </ Wsrp: GetResourceProperty>
      </ SOAP-ENV: Body>
</ SOAP-ENV: Envelope>
```

Рис. 7.19. Запит на отримання інформації про стан ресурсу

8. ТЕХНОЛОГІЇ ГРІД

Термін «грід» уперше використав Ян Фостер на початку 1998 року в книзі «Грід. Нова інфраструктура обчислень» [32]: *«**Грід** – це система, яка*

координує розподілені ресурси з допомогою стандартних, відкритих, універсальних протоколів та інтерфейсів для забезпечення нетривіальної якості обслуговування».

Хоча в останнє десятиліття базова ідея грід не зазнала істотних змін, всеосяжного означення грід досі не існує [33].

8.1. Основні завдання грід

Основна ідея концепції грід-обчислень – централізоване віддалене надання ресурсів, необхідних для вирішення різного роду обчислювальних завдань.

Концепція грід-обчислень у якомусь сенсі перетинається з концепцією електромережі (англ. *Power Grid*): нам не важливо, звідки до нас в розетку надходить електрика, незалежно від цього ми можемо підімкнути до електромережі праску, комп'ютер або пральну машину. Так само і в ідеології грід: ми можемо запустити будь-яке завдання з будь-якого комп'ютера або мобільного пристрою на обчислення, ресурси ж для цього обчислення мають бути автоматично надані на віддалених високопродуктивних серверах незалежно від типу нашої задачі.

Основне завдання грід – узгоджений розподіл ресурсів і вирішення завдань в умовах динамічних багатопрофільних віртуальних організацій.

Розподіл ресурсів – це не просто обмін файлами, а прямий доступ до комп'ютерів, ПЗ, даних та інших ресурсів, які потрібні для спільного вирішення завдань.

Віртуальною організацією (ВО) називають групу окремих людей або установ, об'єднаних єдиними правилами колективного доступу до розподілених обчислювальних ресурсів.

Для організації роботи в межах ВО є необхідними:

- гнучкі механізми розподілу ресурсів;
- розвинена система контролю використовуваних ресурсів;
- розподілений доступ до різних ресурсів, починаючи від програм, файлів і даних і закінчуючи комп'ютерами, сенсорами і мережами;
- різні моделі використання ресурсів (від одного користувача до багатокористувацьких, від високопродуктивних до маловитратних), що містять регулювання якості наданого обслуговування, планування, перерозподіл і ведення обліку ресурсів.

Застосування технологій побудови РВС, що існували на той момент, не давало можливості повною мірою виконати всі зазначені вимоги, тому було запропоновано альтернативну архітектуру грід.

Дослідження і розробки в співтоваристві грід привели до розроблення протоколів, сервісів та інструментарію, спрямованого саме на ті проблеми, які виникають під час створення масштабованих ВО. Ці технології містять:

- ***рішення з безпеки***, що підтримують керування сертифікацією й

політиками безпеки, коли обчислення проводяться декількома організаціями;

- ***протоколи керування ресурсами й сервісами***, що підтримують безпечний віддалений доступ до обчислювальних ресурсів і ресурсів даних, а також перерозподіл різних ресурсів;

- ***протоколи запиту інформації*** і сервіси, що забезпечують налагодження і моніторинг стану ресурсів, організацій і сервісів;

- ***сервіси оброблення даних***, що забезпечують пошук і передання наборів даних між системами зберігання даних і додатками.

8.2. Рівні архітектури грід

Існують такі рівні архітектури грід:

1) ***базовий (Fabric)***, що містить різні ресурси, такі, як комп'ютери, пристрої зберігання, мережі, сенсори й ін.;

2) ***зв'язувальний (Connectivity)***, що визначає комунікаційні протоколи й протоколи аутентифікації;

3) ***ресурсний (Resource)***, що реалізує протоколи взаємодії з ресурсами РВС та їх керування;

4) ***колективний (Collective)***, на якому здійснюються керування каталогами ресурсів, діагностика, моніторинг;

5) ***прикладний (Applications)***, що містить інструментарій для роботи з грід і призначені для користувача програми.

На базовому рівні визначаються служби, які забезпечують безпосередній доступ до ресурсів, використання яких розподілено з допомогою протоколів грід.

1. Обчислювальні ресурси надають користувачеві грід-системи процесорні потужності. Обчислювальними ресурсами можуть бути як кластери, так і окремі робочі станції. Будь-яка обчислювальна система може розглядатися як потенційний обчислювальний ресурс грід-системи.

2. Ресурси пам'яті являють собою простір для зберігання даних. Для доступу до ресурсів пам'яті використовується програмне забезпечення проміжного рівня, що реалізує уніфікований інтерфейс керування і передання даних.

3. Інформаційні ресурси й каталоги є особливим видом ресурсів пам'яті для зберігання й надання метаданих та інформації про інші ресурси грід-системи.

4. Мережний ресурс є сполучною ланкою між розподіленими ресурсами грід-системи. Його основною характеристикою є швидкість передання даних.

Зв'язувальний рівень визначає комунікаційні протоколи й протоколи аутентифікації і забезпечує передання даних між ресурсами базового рівня. Зв'язувальний рівень грід базується на стеку протоколів TCP/IP:

- інтернет (IP, ICMP);

- транспортні протоколи (TCP, UDP);
- прикладні протоколи (DNS, OSRF ...).

Ресурсний рівень реалізує протоколи, що забезпечують виконання таких функцій:

- узгодження політик безпеки використання ресурсу;
- ініціація ресурсу;
- моніторинг стану ресурсу;
- контроль над ресурсом;
- облік використання ресурсу.

Окремо виділяють два типи протоколів ресурсного рівня:

- *інформаційні протоколи*, які використовуються для отримання інформації про структуру і стан ресурсу;
- *протоколи керування*, які застосовуються для узгодження доступу до ресурсів, що розподіляються, визначення вимог і допустимих дій щодо ресурсу (наприклад, підтримка резервування, можливість створення процесів, доступ до даних).

Колективний рівень відповідає за глобальну інтеграцію різних наборів ресурсів і може містити такі служби: каталогів; спільного виділення, планування й розподілу ресурсів; моніторингу й діагностики ресурсів; реплікації даних.

На **прикладному рівні** розташовуються користувацькі додатки, які виконуються в середовищі ВО. Вони можуть використовувати ресурси, що знаходяться на будь-яких нижніх шарах архітектури грід.

8.3. Стандарти грід

Ключовим моментом у розробленні грід-додатків є стандартизація, що дає змогу організувати пошук, використання, розміщення та моніторинг різних компонентів, з яких складається єдина віртуальна система. До початку 2001 року в різних проектах застосовувалися різні методи реалізації грід-обчислень, але спільним для них було те, що найбільш придатною для гнучкого, прозорого й надійного надання доступу до обчислювальних ресурсів є сервісно-орієнтована модель [12].

2001 року як базу для створення стандарту архітектури грід-додатків було вибрано технологію веб-сервісів. Вибір був обумовлений двома основними перевагами цієї технології. По-перше, мова опису інтерфейсів веб-сервісів WSDL забезпечує можливість динамічного пошуку й компонування сервісів у гетерогенних середовищах. По-друге, поширена адаптація механізмів веб-сервісів означає, що інфраструктура, побудована на базі веб-сервісів, може використовувати різні утиліти та інші наявні сервіси.

Розроблений стандарт архітектури грід, що отримав назву OGSA (Open Grid Services Architecture – відкрита архітектура грід-сервісів), ґрунтується на понятті грід-сервісу.

Грід-сервісом називають сервіс, що підтримує надання повної інфо-

рмації про поточний стан екземпляра сервісу, а також можливість надійного і безпечного виконання, керування часом життя, розсилання повідомлень про зміну стану екземпляра сервісу, керування політикою доступу до ресурсів і сертифікатами доступу, віртуалізації.

Грід-сервіс підтримує кілька стандартних інтерфейсів:

- *пошук* – грід-додаткам необхідні механізми для пошуку доступних сервісів і визначення їх характеристик;
- *динамічне створення сервісів* – можливість динамічного створення сервісів та керування ними (це один з базових принципів OGSA, що потребує наявності сервісів створення нових сервісів);
- *керування часом життя* – розподілена система має забезпечувати можливість знищення екземпляра грід-сервісу;
- *повідомлення* – для забезпечення роботи грід-системи додатки й набори грід-сервісів повинні мати можливість асинхронно повідомляти одне одному про зміни їх стану.

Перша реалізація моделі OGSA, розроблена 2003 року, мала назву OGSI (Open Grid Service Infrastructure). У зв'язку з тим, що наявні тоді стандарти веб-сервісів (до яких належали WSDL, SOAP, UDDI) не могли забезпечити всіх вимог, що ставилися розробниками до функціональних можливостей грід-сервісів, при створенні OGSI потрібно модифікувати і розширити відповідні стандарти [34]. Це призвело до того, що спільне використання веб- і грід-сервісів в одному середовищі стало неможливим через несумісність базових стандартів [35].

Подальші спільні зусилля спільноти грід та організацій з розроблення стандартів веб-сервісів призвело до визначення стандартів, що відповідають вимогам грід, зокрема WSRF, у якому специфіковано універсальні механізми для визначення й перегляду стану віддаленого ресурсу та керування ним. Сьогодні реалізація моделі OGSA з допомогою стандарту WSRF (і супутніх стандартів, таких як WS-Notification і WS-Addressing) є найбільш поширеною в середовищі грід.

Нині існують дві системи, що забезпечують інфраструктуру розроблення грід-систем відповідно до стандартів OGSA, реалізованих з допомогою WSRF: Globus [12] і UNICORE [36].

8.4. Система Globus

Globus – це проект з розроблення й надання інфраструктури для грід-обчислень. Спочатку Globus був продовженням проекту I-WAY, але в процесі розвитку основний акцент був перенесений з підтримки високопродуктивних обчислень на підтримку віртуальних організацій.

Мета створення Globus – надання можливості програмам працювати з розподіленими різнорідними обчислювальними ресурсами як з єдиною віртуальною машиною. Основна спрямованість проекту – обчислюва-

льні грід-системи.

Під обчислювальною грід-системою розуміють інфраструктуру апаратних і програмних ресурсів, що реалізує надійний і повномасштабний доступ до високопродуктивних обчислювальних систем незалежно від географічного розташування користувачів або ресурсів.

Базовим елементом системи є Globus Toolkit (інструментарій Globus), що описує базові сервіси й можливості, необхідні для створення обчислювальних грід-систем. Система Globus надає високорівневим додаткам доступ до сервісів, кожен з яких може використовуватися додатком або розробником для досягнення певних цілей, тому окремі сервіси повинні бути ізольованими і мати чітко визначені програмні інтерфейси.

Базові сервіси, що надаються системою Globus, і схему їх взаємодії зображено на рис. 8.1.

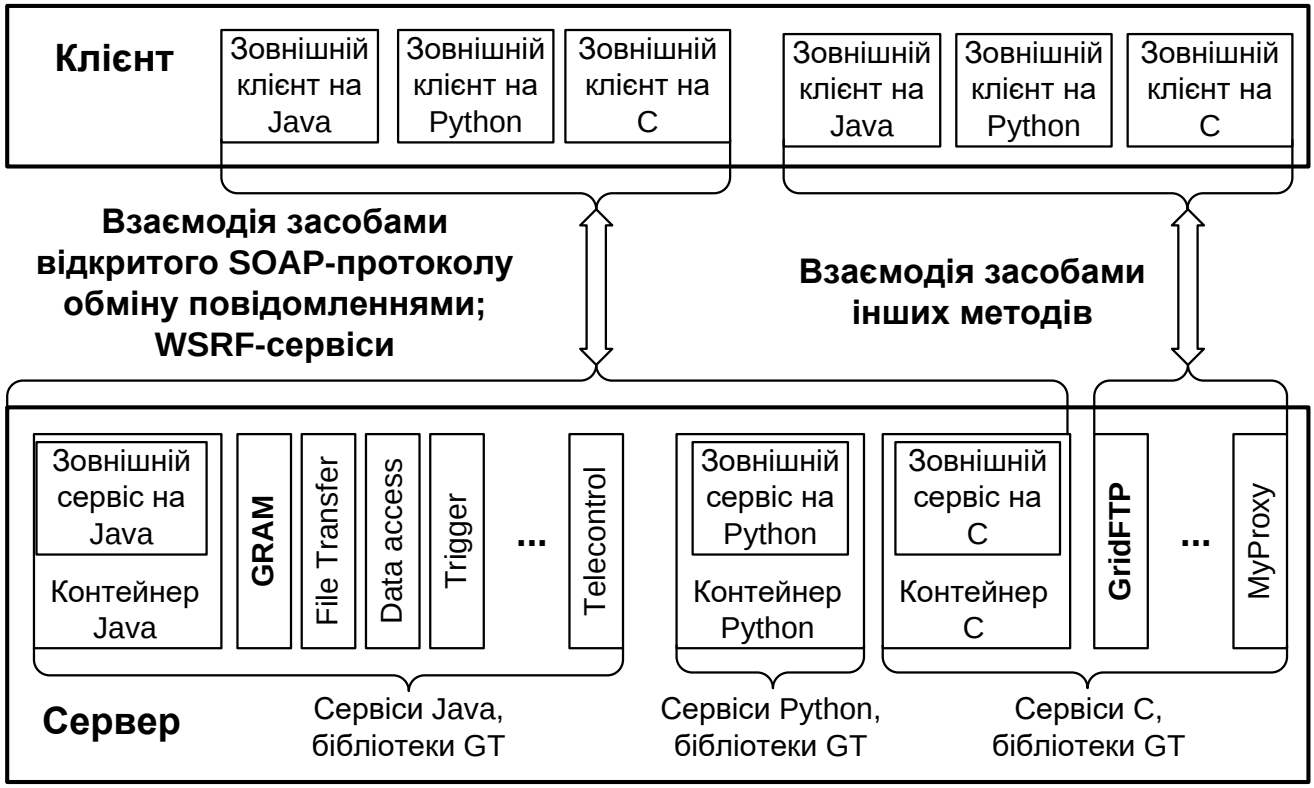

Рис. 8.1. Загальна схема взаємодії компонентів Globus Toolkit 4.0

Сьогодні час до цих сервісів належать:

1. Протокол GRAM (Globus Toolkit Resource Allocation Manager – менеджер розподілу ресурсів Globus Toolkit), який застосовується для розподілу обчислювальних ресурсів і контролю обчислень з використанням цих ресурсів.

2. Розширена версія протоколу передання файлів GridFTP, що використовується для організації доступу до даних, включаючи питання безпеки й паралелізму високошвидкісного передання даних.

3. Контейнери для призначених для користувача сервісів, що підтримують аутентифікацію, керування станом, пошук, а також забезпечують підтримку стандартів WSRF, WS-Security, WS-Notification.

4. Сервіси аутентифікації і безпеки з'єднань GSI (Grid Security Infrastructure – інфраструктура безпеки грід).

5. Розподілений доступ до інформації про структуру і стан системи розподілених обчислень.

6. Віддалений доступ до даних з допомогою послідовних і паралельних інтерфейсів.

7. Можливість створення, кешування й пошуку виконуваних ресурсів.

8. Бібліотеки, які використовуються для забезпечення взаємодії сторонніх додатків і/або призначених для користувача сервісів з GTK 4.0.

8.5. Система UNICORE

Проект UNICORE (Uniform Interface to Computing Resources – єдиний інтерфейс до обчислювальних ресурсів) почали розробляти 1997 року. Нині проект являє собою комплексне рішення, орієнтоване на забезпечення прозорого безпечного доступу до ресурсів грід.

Архітектура UNICORE 6, зображена на рис. 8.2, формується з клієнтського, сервісного й системного шарів.

Верхнім шаром в архітектурі є клієнтський шар. У ньому розташовуються різні клієнти, що забезпечують взаємодію користувачів з грід-середовищем:

- UCC (Unicore Command Line Client - клієнт командного рядка для UNICORE), що забезпечує інтерфейс командного рядка для постановки завдань та отримання результатів;
- URC (Unicore Rich Client – багатофункціональний клієнт UNICORE), який базується на інтерфейсі середовища Eclipse й надає в графічному вигляді повний набір усіх функціональних можливостей системи UNICORE;
- HiLA (High Level API for Grid Applications – високорівневий програмний інтерфейс для додатків грід), що використовується під час розроблення клієнтів до системи UNICORE;
- портали – доступ користувачів до грід-ресурсів через інтернет з допомогою інтеграції UNICORE і систем інтернет-порталів.

Проміжний сервісний шар містить усі сервіси й компоненти системи UNICORE, що ґрунтуються на стандартах WSRF і SOAP:

- шлюз – це компонент, що забезпечує доступ до вузла UNICORE з допомогою аутентифікації всіх вхідних повідомлень;
- компонент XNJS, що забезпечує керування завданнями й виконання ядра UNICORE 6;

- регістр сервісів, що забезпечує реєстрацію й пошук ресурсів, доступних у грід-середовищі.

На рівні сервісного шару також забезпечується підтримка безпечних з'єднань, авторизації та аутентифікації користувачів.

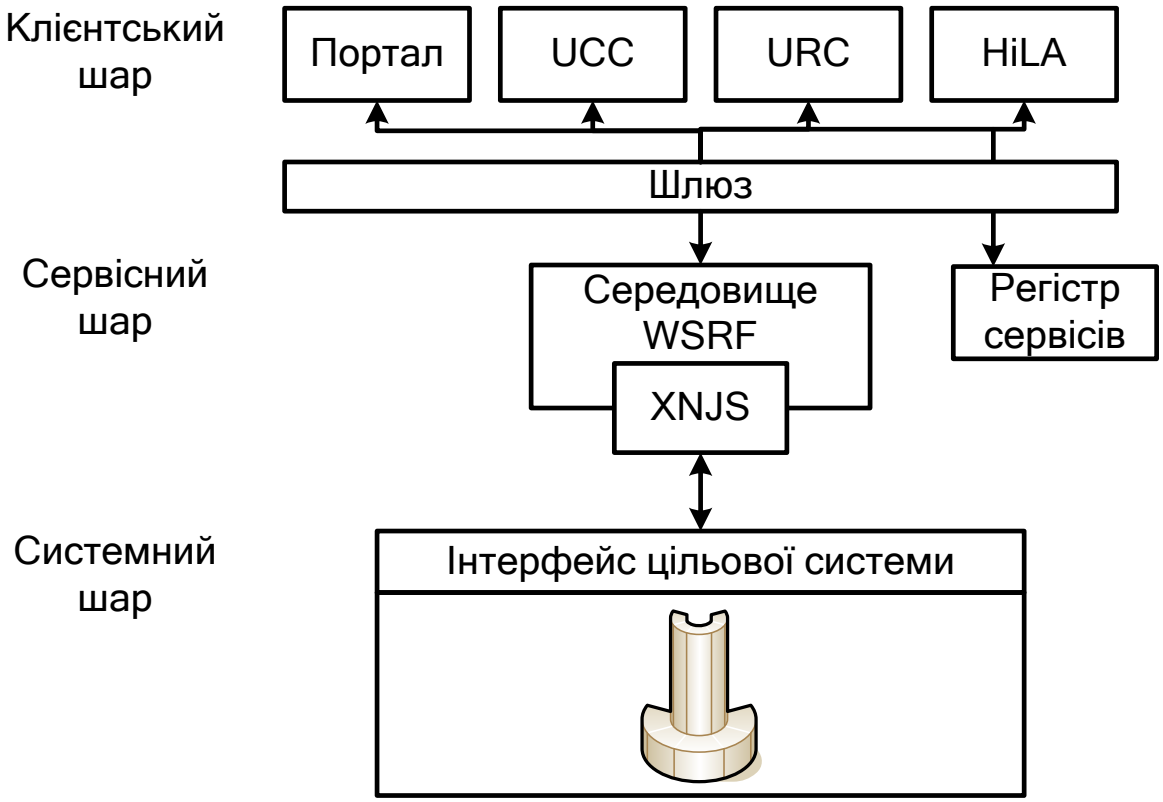

Рис. 8.2. Архітектура системи UNICORE 6

Основою архітектури UNICORE є системний шар. Інтерфейс цільової системи (TSI – Target System Interface) забезпечує взаємодію між UNICORE та окремим ресурсом грід-мережі, трансляцію команд, що надходять з грід-середовища до локальної системи.

Основною перевагою використання системи UNICORE 6 для розроблення РВС є наявність багатого арсеналу різних клієнтів, які здійснюють взаємодію користувача з ресурсами обчислювальної мережі, а також розвинених засобів забезпечення безпеки під час розроблення грід-додатків.

8.6. Параметричні моделі продуктивності грід

Принцип роботи й функціональність грід-додатків значно відрізняються від звичайних послідовних і паралельних систем. Основна відмінність – це можливість агрегування і спільного використання великих наборів гетерогенних ресурсів, розподілених між географічно розділеними областями. До високогетерогенного, розподіленого середовища, що дина-

мічно формується, дуже важко безпосередньо застосувати такі традиційні метрики продуктивності, як швидкість обчислень, пропускна здатність каналу та ін. У зв'язку з цим для оцінювання якості сервісу необхідно застосовувати спеціалізовані метрики [5].

Припустимо, що в грід-середовищі є *m* ресурсів та існує система розподілу завдань *τ*, що забезпечує розподіл поставлених завдань $j \in \tau$ на доступні ресурси. У межах цієї системи кожне завдання може бути розбите на дії $k \in j$**.** Кількість завдань у системі – | *τ* |, кількість дій у завданні – | *j* |. При постановці завдання вказується час d_j, до якого користувач хоче отримати результати.

Кожне завдання *j* і всі його дії $k \in j$ надходять у грід у момент часу r_j. У зв'язку з тим, що грід працює в online-режимі, значення r_j є заздалегідь невідомим для більшості завдань. Як тільки виникає певне завдання, проводиться його планування, після чого здійснюються пошук і виділення ресурсів, необхідних для запуску.

8.6.1. Метрики, що залежать від часу

До метрик продуктивності грід-систем, що залежать від часу, належать:

1. **Мінімально можливий час вирішення j-го завдання**

$$C_j(S) = \max_{k \in j} C_k(S),$$

де Ck (S) – час виконання k-ї ($k \in j$) дії внаслідок фінального розподілу S.

2. **Загальний час вирішення j-го завдання**

$$p_j = C_j(S) - \min_{k \in j}\left(C_k(S) - p_k\right),$$

де p_k – час реалізації k-ї дії.

3. **Показник максимального запізнення завдань**

$$L_{max} = \max_{j \in \tau}\left(C_j(S) - d_j\right).$$

При оптимізації розподіленого середовища намагаються забезпечити $L_{max} \to \min$.

4. **Кількість запізнілих завдань** $TJ\left(j \in \tau \wedge C_j > d_j\right)$.

5. **Величина** споживання ресурсів певною підзадачею $RC_k = p_k m_k$, де m_k – кількість ресурсів, необхідних для виконання k-ї дії. Тоді споживання ресурсів j-м завданням $RC_j = \sum_{k \in j} RC_k$, а споживання ресурсів усіма завданнями планувальника $RC(S) = \sum_{j \in r} RC_j$.

6. **Величина використання доступних ресурсів**, яку розраховують за формулою

$$U = \frac{RC(S)}{m\left(\max\limits_{j\in\tau} C_j(S) - \min\limits_{j\in\tau}\left(C_j(S) - p_j\right)\right)}.$$

7. **Метрика витрат**, що визначається формулою

$$WASTE = Utrue - U.$$

Уведення цієї метрики пов'язане з тим, що в процесі виконання завдання можуть відбуватися збої. У цьому випадку завдання має бути запущене кілька разів для успішного виконання. Унаслідок цього повне споживання ресурсів *RC {k, j} true* і повну величину використання ресурсів *Utrue* можна визначити як відповідну величину плюс витрати на виконання завдань зі збоями. При оптимізації роботи грід-системи ця величина також має бути мінімізована.

8. **Середній час відповіді (Average Response Time – ART) і середній час очікування (Average Wait Time – AWT**), які обчислюють за формулами

$$ART = \frac{1}{|\tau|}\sum_{j\in\tau} C_j(S) \text{ і } AWT = \frac{1}{|\tau|}\sum_{j\in\tau}\left(C_j(S) - p_j\right).$$

Ці метрики є важливими з точки зору користувача грід-системи, оскільки дають змогу оцінити час, що витрачається на виконання його завдання.

9. **Девіація середнього часу очікування**, розрахунок якої дає змогу відносно просто оцінити справедливість використання ресурсів, визначається за формулою

$$AWTD = \frac{1}{|\tau|}\sqrt{\sum_{j\in\tau}\left(C_j(S) - p_j\right)^2 - \left(\sum_{j\in\tau}\frac{C_j(S) - p_j}{|\tau|}\right)^2}.$$

10. **Ефективність грід** (Grid Efficiency – G E)

$$GE = \frac{\sum\limits_{j\in\tau}\left(\left(EndTime_j - StartTime_j\right)\times CPU_{S_j}\times CPUSpeed_j\right)}{\left(EndTime_{lastJob} - SubmitTime_{firstJob}\right)\times\sum\limits_{m\in M}\left(CPU_{S_m}\times CPUSpeed_m\right)}\times 100\%,$$

де $\left(EndTime_{lastJob} - SubmitTime_{firstJob}\right)$ – час роботи системи;

CPU_{S_j} і $CPUSpeed_j$ – кількість процесорів, використаних завданням *j*, та їх продуктивність;

CPU_{S_m} і $CPUSpeed_m$ – кількість процесорів у машині *m* та їх продуктивність.

Метрику ефективності зазвичай використовують для оброблення результатів моніторингу грід-системи.

8.6.2. Метрики, що залежать від обсягу роботи

У сучасних грід-системах успішне завершення виконання певного обсягу роботи може бути навіть більш важливим, ніж прискорення, отримане внаслідок розпаралелювання операцій. У зв'язку з цим означення поняття помилки програми для грід-систем дещо відрізняється від того, яке використовується в інших системах.

Грід-додаток, який не зміг успішно виконатися в межах відведеного йому бюджету, генерує повідомлення про помилку, як тільки виявиться неможливість успішного виконання (наприклад, якщо не знайдено ресурсів для виконання обчислень або у зв'язку з настанням крайнього строку роботи програми).

Відмовостійкість грід-системи – це можливість на якомога більший термін переносити час появи помилки, поки є хоч якісь шанси того, що додаток завершиться успішно. Тому для оцінювання якості роботи грід-системи поряд з метриками, переліченими в попередньому підрозділі, можуть бути використані й такі метрики:

1. **Метрика завершеного обсягу роботи (Workload Completion)** – відношення успішно завершених завдань до обсягу всіх завдань, поставлених планувальником грід-середовища:

$$WC = \sum_{j \in \tau \wedge (j \text{ completed})} 1 \Big/ |\tau|.$$

2. **Метрика завершення дій (Task Completion)** – відношення кількості успішно завершених дій до загальної кількості дій, виконаних у межах системи розподілу завдань:

$$TC = \sum_{j \in \tau \wedge k \in j \wedge (k \text{ completed})} 1 \Big/ \sum_{j \in \tau} |j|.$$

3. **Метрика завершення розблокованих дій (Enabled Task Completion)**

$$ETC = \sum_{j \in \tau \wedge k \in j \wedge (k \text{ completed})} 1 \Big/ \sum_{j \in \tau \wedge k \in j \wedge (k \text{ enabled})} 1.$$

Розблокованою називають дію, яку можна виконати тільки після виконання всіх залежностей для цієї дії.

9. ХМАРНІ ОБЧИСЛЕННЯ

Уперше ідею хмарних обчислень висловив Джозеф Ліклайдер 1970 року. Тоді він відповідав за створення ARPANET. Його ідея полягала в тому, що кожна людина на землі буде підключена до мережі, з якої вона буде отримувати не тільки дані, але й програми.

У той же період інший учений, Джон Маккарті (1927–2011), висловив ідею про те, що обчислювальні потужності будуть надаватися користувачам як послуга (сервіс) [5, 37].

На цьому розвиток хмарних технологій було призупинено аж до 90-х років. Потім хмарні технології стали розвиватися знову, чому посприяли такі фактори [37, 38]:

1. У 90-ті роки значно розширилася пропускна здатність інтернету, і хоча тоді це не дало змоги отримати помітний стрибок у розвитку концепції хмарних обчислень, оскільки компанії і технології того часу ще не були готові до цього, сам факт прискорення інтернету дав поштовх швидкому розвитку хмарних обчислень.

2. Поява 1999 року Salesforce.com – першої компанії, що надала доступ до свого додатка через сайт, що, по суті, було першою реалізацією SaaS (ПЗ як послуга).

3. Розроблення 2002 року компанією Amazon хмарного веб-сервісу, який давав змогу зберігати інформацію й виконувати обчислення.

4. Запуск 2006 року сервісу Amazon під назвою Elastic Compute Cloud (EC2) у вигляді веб-сервісу, який давав змогу його користувачам запускати власні додатки. Сервіси Amazon EC2 і Amazon S3 стали першими доступними сервісами хмарних обчислень.

5. Створення компанією Google платформи Google Apps для веб-додатків у бізнес-секторі.

6. Значну роль у розвитку хмарних технологій відіграли технології віртуалізації, зокрема програмне забезпечення, що дає змогу створювати віртуальну інфраструктуру.

7. Розвиток апаратного забезпечення сприяв не стільки швидкому розвиненню хмарних технологій, скільки їх доступності для малого бізнесу й індивідуальних осіб. Що стосується технічного прогресу, то значну роль у цьому відіграли створення багатоядерних процесорів і збільшення місткості накопичувачів інформації.

9.1. Означення хмарних обчислень та їх особливості

Метафора «хмара» уже давно використовується фахівцями в області мережних технологій для зображення на мережних діаграмах складної обчислювальної інфраструктури (або ж інтернету як такого), що приховує свою внутрішню організацію за певним інтерфейсом. Однак термін «хмарні обчислення» почав використовуватися відносно недавно. Згідно з резуль-

татами аналізу пошукової системи Google термін «хмарні обчислення» («Cloud Computing») став широко застосовуватися наприкінці 2007 р. – на початку 2008 р., поступово витісняючи поняття «грід-обчислення» («Grid Computing»). Однією з перших компаній, що почали вживати цей термін, стала компанія IBM, яка розгорнула на початку 2008 року проект «Blue Cloud» і спонсорувала Європейський проект «Joint Research Initiative for Cloud Computing» [5].

Довгий час термін «хмарні обчислення» не мав усталеного стандартного означення, тому безліч різних корпорацій, учених та аналітиків трактували його по-своєму. При цьому якщо в означеннях, що з'являлися в комерційних виданнях, основна увага приділялася тому, що надається користувачеві в межах концепції хмарних обчислень, то в наукових означеннях наводилися також архітектурні особливості запропонованої технології [5]. Одне з перших означень хмарних обчислень дав Ян Фостер [5]: *«**Хмарні обчислення** – це парадигма великомасштабних розподілених обчислень, що базується на ефекті масштабу, у межах якої пул абстрактних, віртуалізованих, динамічно масштабованих обчислювальних ресурсів, ресурсів зберігання, платформ і сервісів за запитом надається зовнішнім користувачам через інтернет».*

У Національній лабораторії ім. Лоуренса в Берклі дали таке означення хмарних обчислень [5]: ***«Хмарні обчислення** – це не тільки додатки, що поставляються як послуги через інтернет, а й апаратні засоби й програмні системи в центрах оброблення даних, які забезпечують надання цих послуг ... ».*

Одне з найбільш цілісних означень дав інженер-дослідник IV Hewlett-Packard Labs Луїс Вакуеро 2009 року [39]: *«**Хмара** – це великий пул легко використовуваних і легкодоступних віртуалізованих ресурсів (таких, як апаратні комплекси, сервіси та ін.). Ці ресурси можуть бути динамічно перерозподілені (масштабовані) для налагодження під навантаження, що динамічно змінюється, із забезпеченням оптимального використання ресурсів. Цей пул ресурсів зазвичай надається за принципом «оплата у міру використання». При цьому власник хмари гарантує якість обслуговування на основі певних угод з користувачем».*

Нарешті, 2011 року в Національному інституті стандартів і технологій (НІСТ) США сформульовано означення хмарних обчислень, що сьогодні вважається офіційним [40]: *«**Хмарні обчислення** – інформаційно-технологічна концепція, що передбачає забезпечення повсюдного й зручного мережного доступу на вимогу до загального пулу обчислювальних ресурсів, що конфігуруються (наприклад, мереж передачі даних, серверів, пристроїв зберігання даних, додатків і сервісів – як разом, так і окремо), які можуть бути оперативно надані та звільнені з мінімальними експлуатаційними витратами або зверненнями до провайдера».*

У цьому ж документі [40] визначено й обов'язкові характеристики хмарних обчислень:

- самообслуговування на вимогу – споживач самостійно визначає і змінює обчислювальні потреби, такі як серверний час, швидкість доступу й оброблення даних, обсяг збережених даних без взаємодії з представником постачальника послуг;

- ***універсальний доступ по мережі*** – послуги є доступними споживачам через мережу передання даних незалежно від використовуваного термінального пристрою;

- ***об'єднання ресурсів*** – постачальник послуг об'єднує ресурси для обслуговування великої кількості споживачів в єдиний пул для динамічного перерозподілу потужностей між споживачами в умовах постійного змінення попиту на потужності;

- еластичність – послуги можуть бути надані, розширені, звужені в будь-який момент часу без додаткових витрат на взаємодію з постачальником зазвичай в автоматичному режимі;

- ***облік споживання*** – постачальник послуг автоматично обчислює спожиті ресурси на певному рівні абстракції (наприклад, обсяг збережених даних, пропускну здатність, кількість користувачів і транзакцій) і на основі цих даних оцінює обсяг послуг, наданих споживачам.

Хмарні обчислення мають кілька особливостей, основні з яких показано на рис. 9.1.

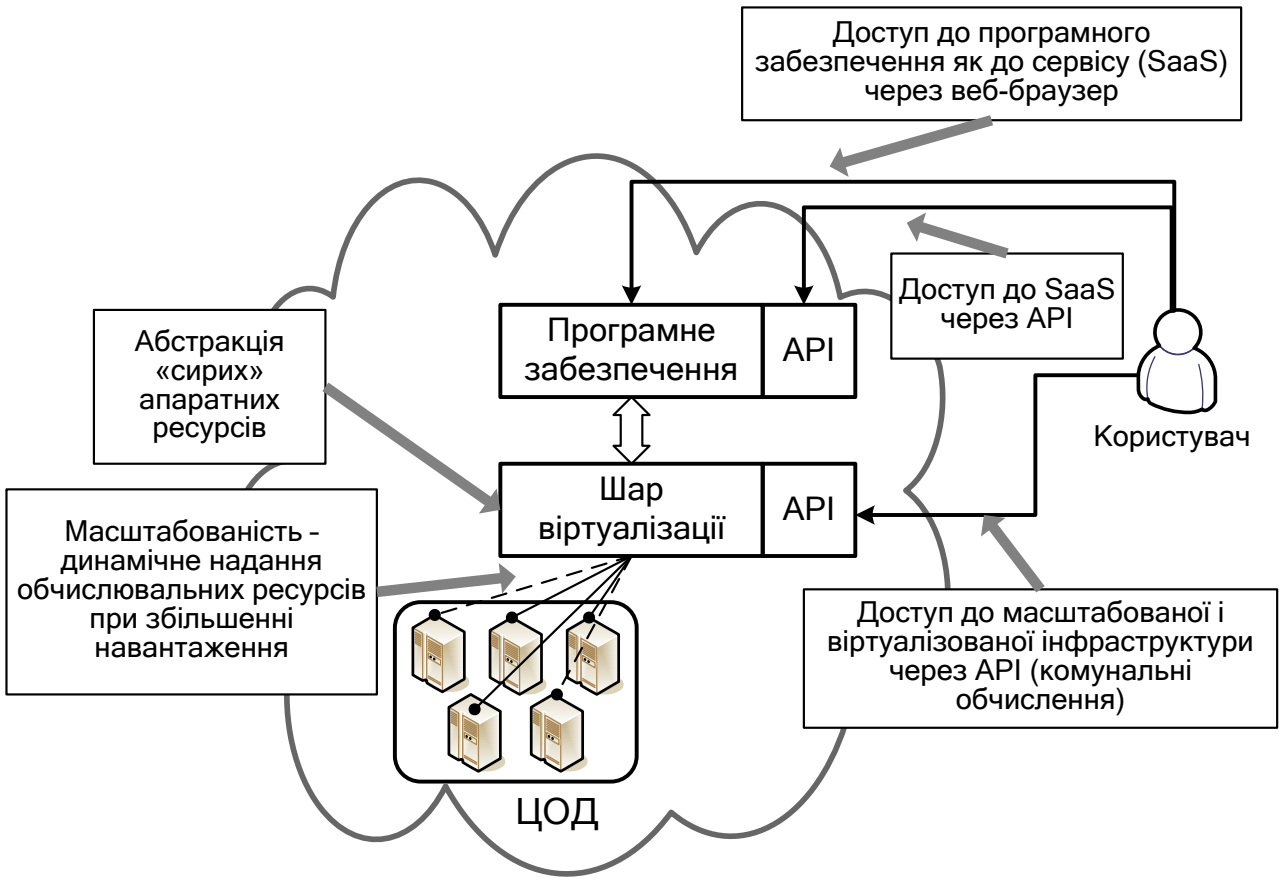

Рис. 9.1. Особливості хмарних обчислень

Як видно, з боку власника обчислювальних ресурсів хмарні обчислення орієнтовані на надання інформаційних ресурсів зовнішнім користувачам, а з боку користувача хмарні обчислення – це отримання інформаційних ресурсів у вигляді послуги у зовнішнього постачальника, оплата за яку проводиться залежно від обсягу спожитих ресурсів відповідно до встановленого тарифу.

Ключовими характеристиками хмарних обчислень є масштабованість і віртуалізація.

Масштабованість – можливість динамічного налагодження інформаційних ресурсів до навантаження, що змінюється, наприклад, до збільшення або зменшення кількості користувачів, змінення необхідної місткості сховищ даних або обчислювальної потужності.

Віртуалізація в основному використовується для забезпечення абстракції й інкапсуляції.

Абстракція дає змогу уніфікувати «сирі» обчислювальні, комунікаційні ресурси і сховища інформації у вигляді пулу ресурсів і вибудувати уніфікований шар ресурсів, який містить ті ж ресурси, але в абстрагованому вигляді. Вони подаються користувачам і верхнім шарам хмарних систем як віртуалізовані сервери, кластери серверів, файлові системи і СКБД.

Інкапсуляція додатків підвищує безпеку, керованість та ізольованість. Ще однією важливою особливістю хмарних платформ є інтеграція апаратних ресурсів і системного ПЗ з додатками, які надаються кінцевому користувачеві у вигляді сервісів.

9.2. Моделі розгортання хмарних систем

Приватна хмара (Private cloud) – інфраструктура, призначена для використання однією організацією, що містить кілька споживачів (наприклад, підрозділів однієї організації), а також клієнтами і підрядниками цієї організації. Приватна хмара може перебувати у власності, керуванні й експлуатації як самої організації, так і третьої сторони (або будь-якої їх комбінації) і фізично існувати як усередині, так і поза юрисдикцією власника.

Публічна хмара (Public cloud) – інфраструктура, призначена для вільного використання широкою публікою. Публічна хмара може перебувати у власності, керуванні й експлуатації комерційних, наукових та урядових організацій (або будь-якої їх комбінації). Публічна хмара фізично існує в юрисдикції власника – постачальника послуг.

Громадська хмара (Community cloud) – інфраструктура, призначена для використання конкретною спільнотою споживачів з організацій, що мають спільні завдання. Громадська хмара може перебувати в кооперативній (спільній) власності, керуванні й експлуатації спільноти споживачів з однієї або більше організацій або третьої сторони (або будь-якої їх

комбінації) і фізично існувати як усередині, так і поза юрисдикцією власника.

Гібридна хмара (Hybrid cloud) – це комбінація з двох або більше різних хмарних інфраструктур (приватних, публічних або громадських), що є унікальними об'єктами, але є пов'язаними між собою стандартизованими або приватними технологіями передання даних і додатків (наприклад, короткочасне використання ресурсів публічних хмар для балансування навантаження між хмарами) [5, 40].

9.3. Моделі обслуговування в хмарних системах

Усі можливі методи класифікації хмар можна звести до тришарової архітектури хмарних систем, що складається з таких рівнів (рис. 9.2) [5]:

- інфраструктура як сервіс (Infrastructure as a Service – IaaS);
- платформа як сервіс (Platform as a Service – PaaS);
- програмне забезпечення як сервіс (Software as a Service – SaaS).

SaaS (Software-as-a-Service – програмне забезпечення як послуга) – модель, у якій споживачеві надається можливість використання прикладного програмного забезпечення провайдера, який працює в хмарній інфраструктурі і є доступним з різних клієнтських пристроїв з допомогою:

- тонкого клієнта (наприклад, з браузера (веб-пошта));
- інтерфейсу програми.

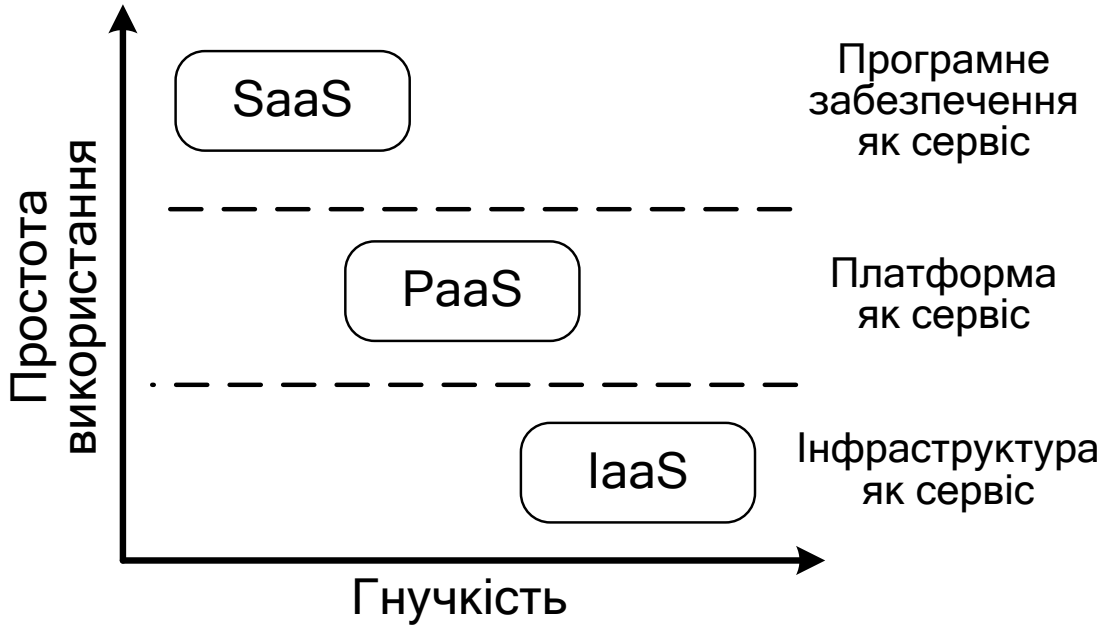

Рис. 9.2. Основні моделі розгортання хмарних систем

Контроль за фізичною і віртуальною інфраструктурою хмари й керування нею (у тому числі мережею, серверами, операційними системами, системами зберігання або навіть індивідуальних можливостей додатка, за

винятком обмеженого набору призначених для користувача налагоджень конфігурації програми) здійснюються хмарним провайдером.

З точки зору користувача, основною перевагою SaaS є нижча вартість порівняно з «класичним» ПЗ. Оплата SaaS здійснюється за моделлю «оплата у міру використання», що означає відсутність необхідності інвестицій у власну апаратну й програмну інфраструктуру.

Яскравим прикладом SaaS є комплекс Google Apps, що містить такі системи, як Google Mail і Google Docs.

PaaS (Platform-as-a-Service – платформа як послуга) – модель, коли споживачеві надається можливість використання хмарної інфраструктури для розміщення базового ПЗ з метою подальшого розміщення на ньому нових або наявних додатків (власних, розроблених на замовлення або придбаних тиражованих). До складу таких платформ входять інструментальні засоби створення, тестування й виконання прикладного ПЗ (системи керування базами даних, сполучне ПЗ, середовища виконання мов програмування), що надаються хмарним провайдером.

Контроль за фізичною й віртуальною інфраструктурою хмари і керування нею, в тому числі мережею, серверами, операційними системами, системами зберігання, за винятком розроблених або встановлених додатків, а також, якщо можливо, параметрами конфігурації середовища (платформи), здійснюються хмарним провайдером.

Платформа – це шар абстракції між програмними додатками (SaaS) і віртуалізованою інфраструктурою (IaaS). Основна цільова аудиторія PaaS – розробники додатків.

Прикладом реалізації PaaS є платформа Google App Engine, що забезпечує виконання призначених для користувача додатків на інфраструктурі Google.

IaaS (Infrastructure-as-a-Service – інфраструктура як послуга) – модель, у якій надається можливість використання хмарної інфраструктури для самостійного керування ресурсами оброблення, зберігання, мережами та іншими фундаментальними обчислювальними ресурсами. Наприклад, споживач може встановлювати й запускати довільне програмне забезпечення, що складається з операційної системи, платформного і прикладного ПЗ, контролювати операційні системи, віртуальні системи зберігання даних і встановлені програми, а також здійснювати обмежений контроль за набором доступних мережних сервісів (наприклад, фаєрволом, DNS).

Контроль за фізичною і віртуальною інфраструктурою хмари і керування нею, в тому числі мережею, серверами, типами використовуваних операційних систем, системами зберігання, здійснюються хмарним провайдером.

Яскравим прикладом такого підходу є хмара компанії Amazon – Amazon Web Services, що складається з Elastic Compute Cloud (EC2), що

надає інформаційні ресурси у вигляді сервісів, і Simple Storage Service (S3), що використовується для зберігання інформації.

Слід зазначити, що задовго до появи хмарних обчислень інфраструктура була доступною як сервіс. Такий підхід мав назву «комунальні обчислення». Це словосполучення й сьогодні часто застосовують деякі автори під час опису інфраструктурного рівня хмарних систем.

У деяких джерелах виділяють й інші моделі обслуговування в хмарних системах [38], наприклад:

- **апаратне забезпечення як послуга (Hardware as a Service – HaaS)**: користувач має обладнання на правах оренди, яке він може використовувати для власних цілей; по суті, HaaS нагадує IaaS, за винятком того, що є «голе» обладнання, на основі якого можна розгорнути свою власну інфраструктуру з використанням найбільш придатного ПЗ; перевагою HaaS є можливість економити на обслуговуванні обладнання;

- **робоче місце як послуга (Workplace as a Service – WaaS)**: компанія використовує хмарні обчислення для організації робочих місць своїх співробітників, налаштувавши і встановивши все необхідне для роботи персоналу ПЗ;

- **дані як послуга (Data as a Service – DaaS)**: користувачеві надається дисковий простір, який він може використовувати для зберігання великих обсягів інформації;

- **безпека як послуга (Security as a Service)**: користувачам надається можливість швидко розгортати продукти, що забезпечують безпечне використання веб-технологій, електронного листування, а також локальної системи, що дає змогу економити на утриманні власної системи безпеки;

- **усе як послуга (Everything as a Service – EaaS)**: користувачеві надається можливість керувати як програмно-апаратною частиною, так і бізнес-процесами, включаючи взаємодію між користувачами, за наявності доступу до мережі інтернет; EaaS – це просто більш загальне поняття щодо перелічених вище послуг.

9.4. Компоненти хмарних додатків

Через високу закритість різних аспектів реалізації найбільш поширених хмарних систем сьогодні не існує єдиної компонентної архітектури хмарних додатків. Проте можна виділити основні компоненти, властиві практично всім наявним хмарним платформам (рис. 9.3).

Платформа є центральним компонентом моделі хмари.

Платформа – середовище та набір утиліт, що забезпечують розроблення, інтеграцію й надання хмарних сервісів.

Особливості платформ:

- надання набору базових сервісів, доступних розробнику хмарного додатка;

- накладання певних обмежень на методи розроблення і подання додатка.

При виборі платформи можна як ґрунтуватися на вже готових рішеннях, так і самостійно розробити масштабовану платформу на базі готової хмарної інфраструктури. Основними критеріями вибору базової платформи є вартість закінченого рішення, продуктивність і необхідна масштабованість. При цьому слід пам'ятати, що будь-яка платформа потребує використання певних мов програмування і програмних фреймворків для реалізації програми.

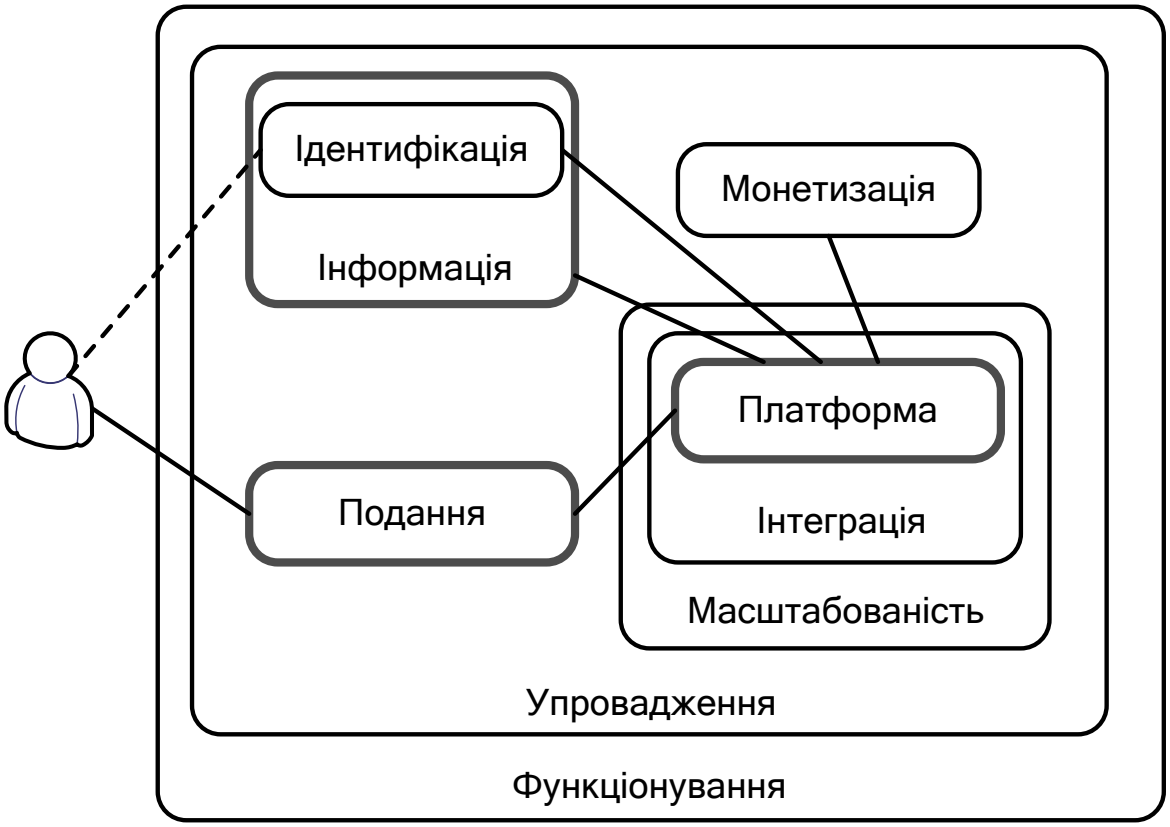

Рис. 9.3. Компоненти хмарних додатків

Подання – це інтерфейс, через який користувач взаємодіє з хмарою. Цей компонент забезпечує отримання вхідних даних і надання інформації кінцевому користувачу.

Найбільш типовим методом реалізації подання є веб-додаток, що забезпечує взаємодію з користувачем за допомогою веб-браузера, хоча останнім часом широко використовуються окремі інтерфейси для мобільних пристроїв (смартфонів, планшетів) з метою забезпечення на цих пристроях максимально повної функціональності.

Інформація – це джерела даних, що забезпечують розподілене зберігання структурованих або неструктурованих, статичних або динамічно змінних даних. Призначена для користувача інформація в хмарних системах може набирати величезних обсягів, на яких класичні SQL-бази да-

них уже не дають задовільних результатів щодо швидкості оброблення. У зв'язку з цим в останні кілька років стали активно розвиватися альтернативні NoSQL-системи керування базами даних та альтернативні підходи до оброблення надвеликих обсягів інформації.

Інтеграція – інфраструктура, яка спрощує обмін інформацією й виконання завдань у розподіленому обчислювальному середовищі. У межах цього компонента необхідно забезпечити максимальні продуктивність і безпеку процесу обміну даними між сервісами, сумісність форматів даних і розробити механізми синхронної й асинхронної взаємодій з успадкованим ПЗ. На більш високому рівні слід забезпечити слабозв'язаність програмних компонентів і переконатися у відсутності вузьких місць у програмній архітектурі системи.

Масштабованість – гнучкість методів надання ресурсів, що забезпечує підтримку виділення додаткових інформаційних ресурсів при збільшенні навантаження на додаток. При цьому необхідно не тільки враховувати можливість короткочасного збільшення навантаження на додаток, а й планувати довгострокове збільшення продуктивності системи внаслідок постійного приросту аудиторії. В обох випадках слід забезпечити декомпозицію хмарного додатка на окремі модульні компоненти, які можуть бути розподілені на кілька обчислювальних пристроїв.

Монетизація – облік і білінг ресурсів, витрачених на виконання користувацьких завдань. Це ключовий компонент безлічі комерційних додатків. Для здійснення якісного білінгу хмарних платформ необхідно організувати збір і надання повноцінної інформації про всілякі ресурси, що витрачаються на вирішення завдань користувача, і забезпечити користувачеві можливість зручної та швидкої оплати витрачених ресурсів.

Упровадження – процес створення нового хмарного додатка, що полягає в його розробленні, тестуванні й введенні в експлуатацію. Застосування готової хмарної інфраструктури дає змогу значно зменшити витрати на розроблення і впровадження високомасштабованого додатка, оскільки оплата використаних інформаційних ресурсів проводиться на основі моделі комунальних обчислень і не потребує значних інвестицій у власну інфраструктуру. Це дає змогу мінімізувати початкові витрати й сконцентрувати фінансування на всебічному тестуванні програми.

Функціонування – моніторинг і підтримка додатків, що перебувають на стадії експлуатації.

Додаток, який запущено в експлуатацію, необхідно адмініструвати, що може бути надзвичайно складним завданням, якщо врахувати велику кількість окремих сервісів, з яких складається хмарний додаток. У зв'язку з цим слід забезпечити інтеграцію процесів адміністрування й керування сервісами у вигляді єдиного «центру керування сервісами», до складу якого також можна включити моніторинг навантаження додатка, панель керування призначеними для користувача завданнями і т. ін.

Усі розглянуті компоненти хмарного додатка мають бути заплановані з самого початку його розроблення для забезпечення високого рівня масштабованості й автоматизації.

9.5. Переваги й недоліки хмарних обчислень

Існують три групи споживачів хмарних обчислень:

- кінцеві користувачі, які використовують SaaS-рішення через веб-браузер або ж будь-які базові ресурси інфраструктурного шару, що надаються з допомогою шару SaaS;

- корпоративні споживачі, які можуть використовувати всі три шари: IaaS – для того, щоб розширити власну програмно-апаратну інфраструктуру або отримати додаткові обчислювальні ресурси на вимогу; PaaS – для того, щоб мати можливість запускати власні додатки в хмарі; SaaS – для отримання можливостей тих додатків, що вже є доступними в хмарі;

- розробники й незалежні постачальники програмного забезпечення, які розробляють додатки, що надаються у вигляді хмарних SaaS-рішень; зазвичай ця категорія користувачів безпосередньо взаємодіє з шаром PaaS і вже через нього, опосередковано, з шаром IaaS.

Основні переваги хмарних обчислень:

1. **Доступність.** Хмари є доступними всім з будь-якої точки, де є інтернет, із будь-якого комп'ютера, де є браузер. Це дає змогу користувачам (підприємствам) економити на закупівлі високопродуктивних дорогих комп'ютерів. Співробітники компаній також стають більш мобільними, оскільки можуть отримати доступ до свого робочого місця з будь-якої точки земної кулі, використовуючи ноутбук, нетбук, планшет або смартфон. Немає необхідності в покупці ліцензійного ПЗ, його налагодженні й оновленні, можна просто зайти на сервіс і користуватися його послугами, заплативши за фактичне використання.

2. **Низька вартість.** Основні фактори, що знизили вартість використання хмар:

- розвиток технологій віртуалізації, що сприяло зниженню витрат на обслуговування віртуальної інфраструктури, унаслідок чого стало можливим зменшення штату для обслуговування всієї IT-інфраструктури підприємства;

- оплата фактичного використання ресурсів, що дає змогу користувачам (підприємствам) економити на покупці ліцензій на ПЗ;

- використання хмари на правах оренди, що дає можливість користувачам знизити витрати на закупівлю дорогого устаткування й направити кошти на налагодження бізнес-процесів підприємства;

- розвиток апаратної частини обчислювальних систем, що сприяло зниженню вартості обладнання.

3. **Гнучкість** – необмеженість обчислювальних ресурсів (пам'яті, процесора, дисків). Завдяки використанню систем віртуалізації процес масштабування й адміністрування хмар стає досить легким завданням, оскільки хмара самостійно може надати необхідні ресурси, потрібно тільки заплатити за їх фактичне використання.

4. **Надійність.** «Хмари», особливо ті, що знаходяться в спеціально обладнаних ЦОД, є дуже надійними, оскільки ЦОД мають резервні джерела живлення, охорону, професійних працівників, регулярне резервування даних, високі пропускну здатність інтернет-каналу і стійкість до DDOS-атак.

5. **Безпека.** Хмарні сервіси мають досить високу безпеку при належному її забезпеченні, однак при недбалому ставленні ефект може бути повністю протилежним.

6. **Великі обчислювальні потужності.** Користувач хмарної системи може використовувати всі її обчислювальні можливості, заплативши тільки за фактичний час їх використання. Ця можливість є особливо цікавою для підприємств, що здійснюють аналіз великих обсягів даних.

Незважаючи на очевидні переваги є і кілька недоліків:

1. **Постійне з'єднання з мережею.** Для отримання доступу до послуг хмари необхідним є постійне з'єднання з мережею інтернет.

2. **ПЗ та його кастомізація.** Є обмеження щодо ПЗ, яке можна розгортати на хмарах і надавати користувачеві. Крім того, через існуючі обмеження користувач іноді не має можливості налагодити надане ПЗ під власні цілі.

3. **Конфіденційність** даних, що зберігаються на публічних хмарах. Нині це спричиняє багато суперечок, але в більшості випадків експерти сходяться в тому, що не слід зберігати найбільш цінні для компанії документи на публічній хмарі, оскільки поки немає технології, яка б гарантувала 100%-кову конфіденційність даних, що зберігаються.

4. **Надійність.** Що стосується надійності зберігання інформації, то з упевненістю можна сказати, що якщо інформація, яка зберігається в хмарі, втрачається, то вона втрачається назавжди.

5. **Безпека.** Хмара сама по собі є досить надійною системою, однак при проникненні на неї зловмисник отримує доступ до величезного сховища даних. Існує ще один недолік – це використання систем віртуалізації, у яких як гіпервізор застосовуються ядра стандартних ОС, таких як Linux, Windows та ін., що дає змогу використовувати віруси.

6. **Висока вартість обладнання.** Для побудови власної хмари компанії необхідно виділити значні матеріальні ресурси, а це не вигідно для щойно створених і малих компаній.

9.6. Найбільш поширені хмарні платформи

Розглянемо можливості й організацію найбільш поширених сьогодні платформ хмарних обчислень: Amazon Web Services, Google App Engine і Microsoft Windows Azure. Їх основні характеристики наведено в табл. 9.1.

Таблиця 9.1

Порівняльні характеристики платформ хмарних обчислень

Характеристика	Платформа		
	Amazon Web Services	Google App Engine	Microsoft Windows Azure
Тип	IaaS	PaaS	PaaS
Сервіси, що розробляються	Обчислювальні послуги, послуги зберігання	Web-додатки	Web-програми, але не Web-додатки
Віртуалізація	На рівні ОС, із запущеним гіпервізором Xen	З використанням контейнера додатків	На рівні ОС
Інтерфейс доступу користувача	Утиліти консолі Amazon EC2	Web-консоль адміністрування	Портал Microsoft Windows Azure
Web APIs	Так	Так	Так
Середовище розроблення	Немає	Python, Java	Microsoft .NET

9.6.1. Amazon Web Services

Загальну архітектуру Amazon Web Services (AWS) зображено на рис. 9.4. Ядром Amazon Web Services є система Amazon Elastic Compute Cloud (Amazon EC2) спільно з сервісами зберігання. EC2 дає користувачеві можливість вибору віртуальних машин, які можна запустити в розподіленому обчислювальному середовищі. Віртуальна машина (Amazon Machine Image – AMI) може базуватися практично на будь-якій операційній системі (різні версії Windows і дистрибутиви Linux) і забезпечувати роботу з будь-якою програмною інфраструктурою (MySQL, Oracle та ін.).

Поряд з віртуальними машинами Amazon забезпечує кілька механізмів зберігання даних:

1. **Simple Storage Service (S3)** – сервіс зберігання даних, що надає доступ на основі REST і SOAP. Система S3 розподілена по всьому світу: сервери розташовані в Європі, Азії та США. При цьому забезпечується можливість роботи з даними розміром від 1 байта до 5 Тб.

2. **Система Elastic Block Storage (EBS)** – високопродуктивний віртуальний жорсткий диск розміром від 1 Гб до 1 Тб. Він може бути підключений до будь-якої віртуальної машини, що працює в межах EC2, і поміщений на тривале зберігання в систему S3.

3. **Сервіси реляційних баз даних** – це веб-сервіси, що забезпечують установлення й масштабування реляційних баз даних у хмарі й керування ними. Зараз надається підтримка баз даних на основі MySQL (надалі планується впровадження і баз даних Oracle).

4. **Amazon CloudFront** – веб-сервіс для надання контенту. Забезпечується статичне і потокове передання даних. У зв'язку з розподіленою інфраструктурою системи CloudFront може забезпечити мінімальну латентність надання інформації, вибираючи сервер, географічно найближчий до користувача.

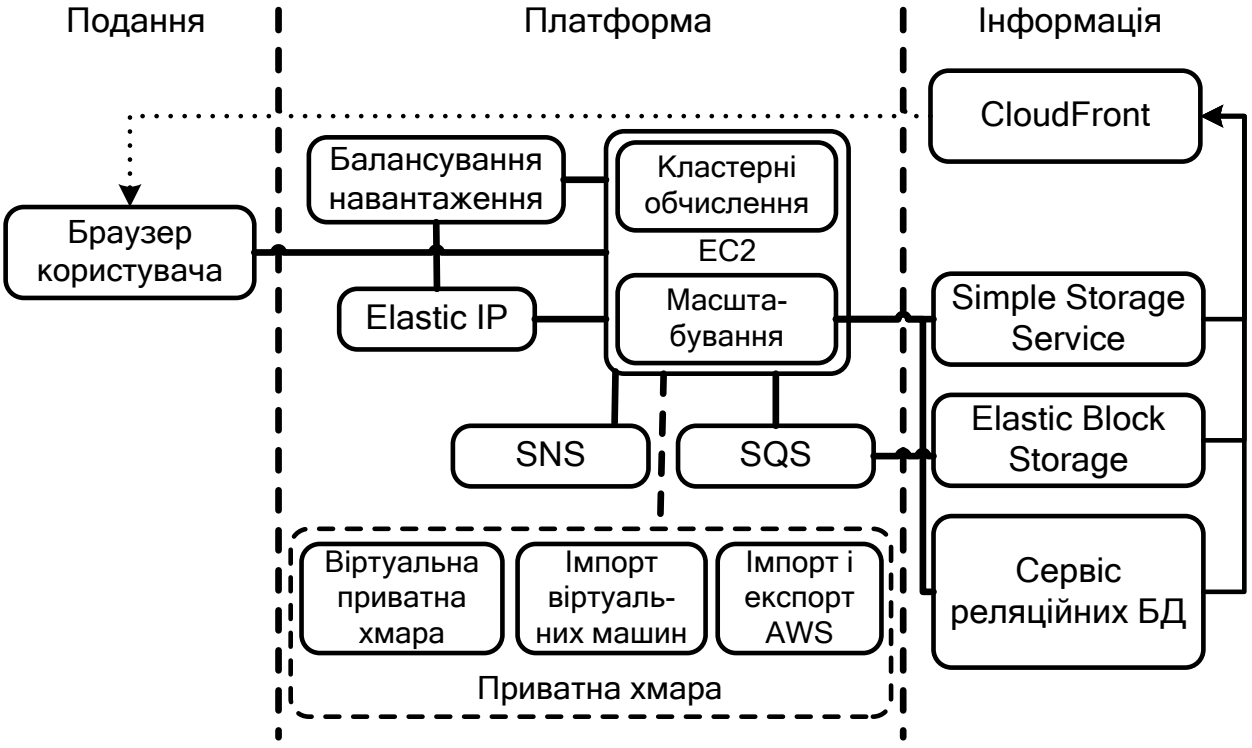

Рис. 9.4. Загальна архітектура Amazon Web Services

Однією з характерних рис платформи Amazon є надання сервісів інтеграції додатків, що забезпечують прозору адресацію й доступність.

Elastic IP забезпечує прив'язування статичної IP-адреси до аккаунту користувача. При цьому в разі помилок і збоїв у роботі окремих віртуальних машин проводиться автоматичне перепризначення IP-адреси іншій віртуальній машині.

Simple Queue Service (SQS) забезпечує розробників практично нескінченною кількістю черг. Кожний авторизований додаток може реєструватися в черзі, відправляти, отримувати або видаляти повідомлен-

ня. При цьому неприйняті повідомлення можуть залишатися в системі аж до чотирьох днів.

Simple Notification Service (SNS) – це сервіс, що забезпечує оповіщення про зміну стану за передплатою. Користувачі системи, хмарні програми та пристрої можуть відправляти й отримувати повідомлення з хмари.

Сьогодні Amazon виходить на корпоративний ринок, надаючи послуги зі створення віртуальних приватних хмар, що розширюють можливості корпоративних обчислювальних інфраструктур. По суті, пропонується організація віртуальних приватних мереж (VPN), які гарантують безпечний і прозорий зв'язок між внутрішньою корпоративною мережею та EC2. Забезпечується імпорт корпоративних віртуальних машин, а також можливість пересилання віртуальних машин на фізичних носіях, минаючи інтернет (підвищуються надійність і безпека передання даних).

Рішення, запропоноване AWS, забезпечує максимально можливу гнучкість при розробленні та впровадженні хмарних рішень. Найбільш активними користувачами цієї платформи є великі організації або проекти, для яких необхідними є максимальна масштабованість і гнучкість розроблення. При цьому таке рішення може виявитися надмірно складним і навантаженим для окремих розробників або додатків, яким не потрібні настільки потужні механізми масштабування.

9.6.2. Google App Engine

Загальну архітектуру Google App Engine (GAE) показано на рис. 9.5.

У GAE користувач отримує доступ до хмарного додатка з допомогою веб-браузера. Нині GAE підтримує розробки на базі мови Python і всіх мов, які можуть виконуватися всередині віртуальної машини Java (Java, Jython, Scala і т. ін.). Розробникам Google App Engine надається комплект засобів розроблення (SDK) для повноцінної симуляції роботи Google App Engine на робочій машині.

GAE надає великий набір бібліотечних функцій для виконання таких стандартних операцій:

- робота з поштовими повідомленнями;
- авторизація й аутентифікація користувачів;
- оброблення зображень;
- завантаження й оброблення веб-сторінок;
- планування завдань;
- оброблення даних за допомогою MapReduce (модель розподілених обчислень, що використовується для паралельного оброблення дуже великих, порядку декількох петабайтів, наборів даних у комп'ютерних кластерах);

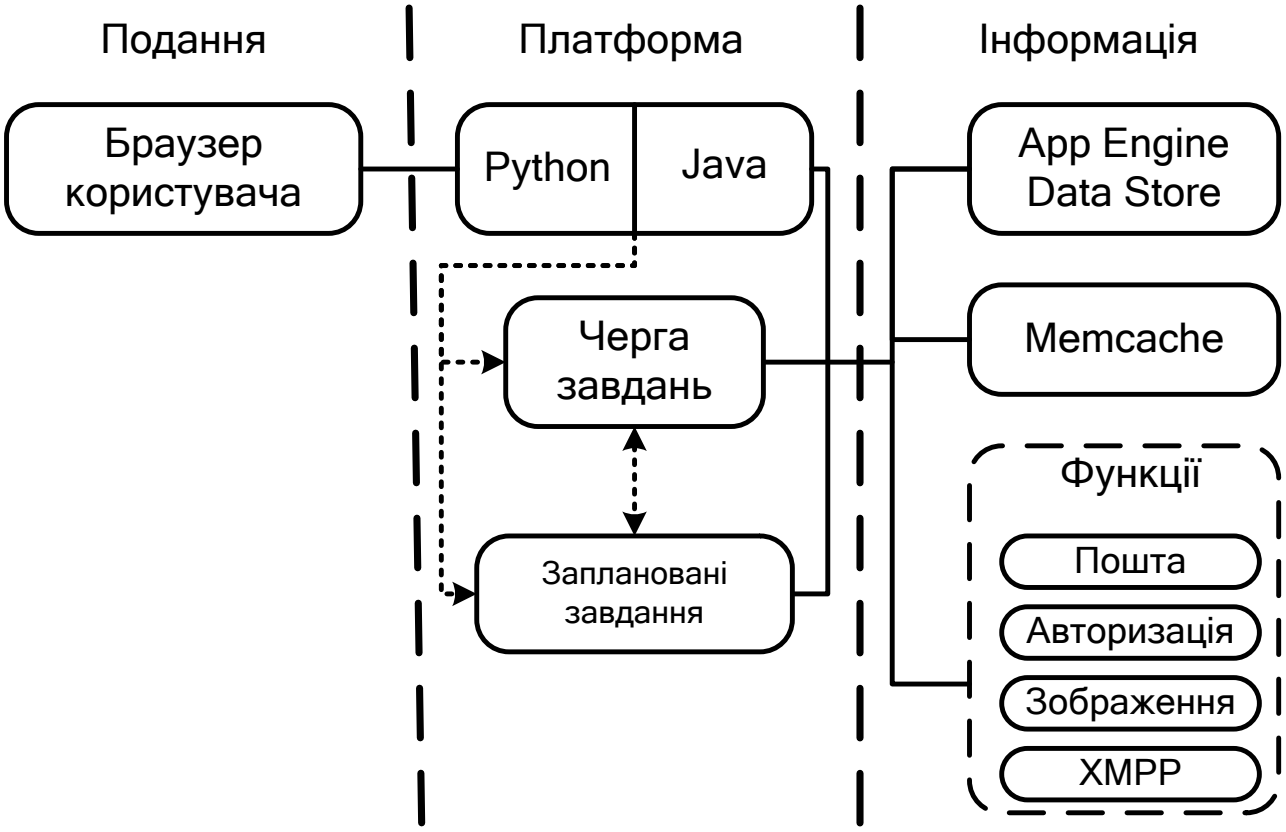

Рис. 9.5. Загальна архітектура Google App Engine

- зберігання великих (до 2 GB) обсягів інформації;
- обмін повідомленнями на основі протоколу XMPP (Jabber) (Extensible Messaging and Presence Protocol – відкритий, вільний для використання протокол для миттєвого обміну повідомленнями та інформацією про присутність у режимі, близькому до реального часу, що базується на XML).

GAE забезпечує можливість надання високопродуктивного сервісу без істотних витрат на створення інфраструктури.

Однак виконання коду в хмарі певною мірою обмежує набір бібліотек, доступних розробнику. Наприклад, сьогодні не підтримуються модулі Python, написані мовою C. Аналогічно Java-додатки можуть використовувати тільки обмежений підклас бібліотек JRE SE і не можуть працювати з потоками. У зв'язку з цим практично неможливо просто взяти і скопіювати готовий Python- або Java-додаток у Google App Engine і сподіватися, що він запуститься в хмарі. Основні труднощі, з якими доведеться стикнутися при перенесенні додатка в хмару Google App Engine, – це взаємодія зі сховищем App Engine Datastore і обмеження доступу до локальних ресурсів у зв'язку з роботою в «пісочниці» віртуальної машини.

App Engine Datastore – це високорозподілена система зберігання даних, яка ґрунтується на пропрієтарній базі даних BigTable, розробленій компанією Google. Методи роботи зі сховищем дуже сильно відрізняються

від методів роботи з реляційними базами даних. Основною відмінністю є те, що в App Engine Datastore немає схем даних. Усі дані зберігаються у вигляді блоків «ключ» – «значення» – «мітка часу». Тому розробник має уважно стежити за відповідністю збережених даних бізнес-логіці програми.

У SDK надаються спеціальні бібліотеки, які даєть змогу забезпечити узгодженість даних. Google розробив власну мову запитів (GQL), схожу на вирази SELECT у мові SQL, але з великою кількістю обмежень.

Віртуальна машина GAE має обмежений доступ до локальних ресурсів, тому під час роботи з нею розробник виявляється як би замкнутим у межах «пісочниці», зокрема, значно обмежено набір протоколів і портів, через які додаток може зв'язуватися із зовнішнім світом (практично залишена можливість зв'язку тільки за допомогою HTTP і HTTPS) і заборонено операції роботи з локальною файловою системою (дозволено тільки читання файлів, які було завантажено разом з кодом програми).

9.6.3. Microsoft Windows Azure

Загальну архітектуру Microsoft Windows Azure зображено на рис. 9.6. Платформа Windows Azure дає змогу запускати додатки, розроблені Microsoft, забезпечуючи автоматичне керування обчислювальними ресурсами, балансування навантаження й реплікацію даних. Доступ розробника до платформи здійснюється за допомогою інструментарію, інтегрованого в останні версії Visual Studio (Windows Azure SDK, Windows Azure Tools for MVS). При цьому підтримується можливість локального тестування додатків до їх публікації на сервісі Azure.

Платформа Windows Azure складається з трьох основних компонентів:

- обчислювальних сутностей;
- сутностей зберігання;
- фабрик.

Обчислювальні сутності – це контейнери для додатків з підтримкою сучасних технологій розроблення, включаючи .NET, Java, PHP, Python, Ruby on Rails і нативний код.

Обчислювальні сутності можуть відігравати дві основні ролі:

- веб-роль (Web Role) – це веб-додаток, доступний користувачеві через інтернет з допомогою веб-браузера;

- прикладна роль (Worker Role) – це програма, яка виконує деяке обчислювальне навантаження у фоновому режимі.

Сутності зберігання – це набір сервісів, що забезпечують зберігання даних.

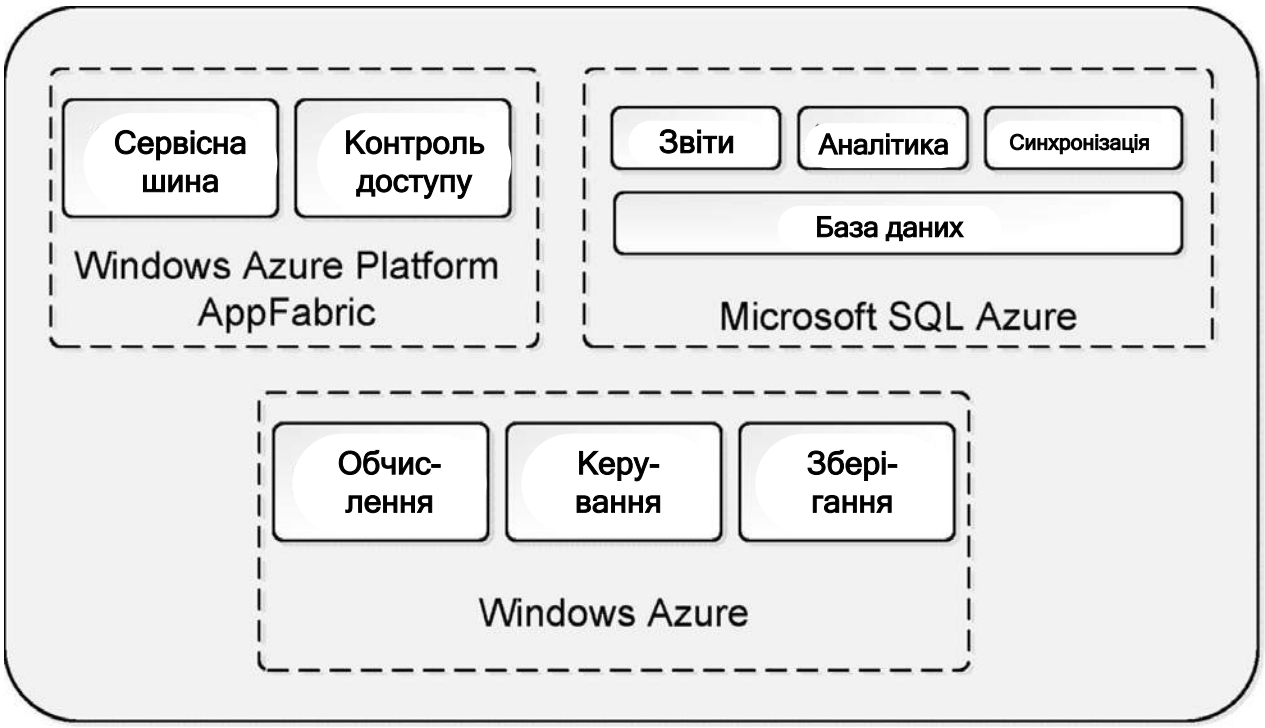

Рис. 9.6. Загальна архітектура Microsoft Windows Azure

Кожен сервіс забезпечує свій тип зберігання даних.

Бінарні об'єкти застосовуються для зберігання великих наборів неструктурованих байтів (файлів), при цьому забезпечується можливість іменування файлів і роботи з метаданими. Максимальний обсяг файлу, що зберігається, становить не більше 1 Тб.

Таблиці використовуються для зберігання структурованих даних і являють собою набір однорідних рядків (так званих сутностей), структура яких визначається набором стовпців (так званих властивостей). Один об'єкт може мати близько 256 властивостей. Таблиці розподілені таким чином, щоб максимально підтримувати балансування навантажень.

Черги – механізм, що забезпечує асинхронну взаємодію додатків з допомогою зберігання й передання повідомлень. Допускається використання необмеженої кількості черг, а черги можуть містити необмежену кількість повідомлень.

Диски. Як і AWS, Windows Azure дає змогу керувати дисками в межах своїх віртуальних машин, а також працювати з томами NTFS, забезпечуючи їх доступність для додатків. Це спрощує процес міграції звичайних додатків у хмару, оскільки з'являється можливість збереження стану у файловій системі.

Windows Azure керується з допомогою спеціального інфраструктурного шару під назвою Windows Azure Fabric Controller (WAFC).

Завдання WAFC – організація масиву віртуальних машин на основі Windows Server у вигляді єдиного віртуального сервера, що забезпечує автоматичне керування ресурсами, балансування навантаження та ін.

Платформа Windows Azure також забезпечує ряд сервісів, доступних розробникам як інтернет-, так і класичних десктопних додатків. Найбільш важливим є компонент SQL Azure, що забезпечує надання реляційної бази даних Microsoft SQL Server як сервісу. Оскільки ця технологія базується на класичному підході SQL Server, вона не забезпечує такої масивної масштабованості, як сутності зберігання (максимальний обсяг інформації становить 10 Гб на одну базу даних). Але, незважаючи на це, не можна недооцінювати всі ті можливості, які можуть бути надані класичною архітектурою, включаючи транзакційну цілісність і аналіз даних. Надаються також служби побудови звітів.

9.7. Порівняння хмарних і грід-обчислень

Навколо питання про те, чим хмарні й грід-обчислення відрізняються одне від одного, точиться безліч дискусій. Однак зазвичай думки сходяться в одному – хмарні обчислення виросли з концепції грід.

Ян Фостер визначає взаємодію грід- і хмарних обчислень у такий спосіб [5]: *«Ми вважаємо, що хмарні обчислення не просто перетинаються з концепцією грід. Насправді хмари виросли з грід-обчислень і ґрунтуються на концепції інфраструктури грід. Еволюція підходу полягає в тому, що замість надання «сирих» обчислювальних ресурсів і ресурсів зберігання забезпечується надання більш абстрактних ресурсів у вигляді сервісів».*

Таким чином, можна вважати що грід- і хмарні обчислення доповнюють одре одного. Грід-обчислення забезпечують об'єднання гетерогенних обчислювальних ресурсів в єдине обчислювальне середовище, забезпечуючи те, з чого починаються і на чому ґрунтуються хмарні обчислення. Хмарні обчислення забезпечують більш високий рівень абстракції, надаючи обчислювальні ресурси кінцевим користувачам (будь то приватні клієнти чи організації) у вигляді сервісів.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Lamport, L. My writings [Electronic Resource] / L. Lamport. – Available from: http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html. – 16.10.2015.
2. Таненбаум, Э. Распределенные системы: принципы и парадигмы / Э. Таненбаум. – СПб. : Питер, 2003. – 877 с.
3. Родин, А. В. Параллельные или распределенные вычислительные системы / А. В. Родин, В. Л. Бурцев // Компьютерные системы и технологии. – М. : МИФИ, 2006. – Т. 12. – С. 151–153.
4. Grid Computing in Research and Education [Electronic Resource] / L. Ferreira, F. Lucchese, T. Yasuda, C. Y. Lee, C. A. Queiroz, E. Minetto, A. Mungioli. – IBM : RedBooks, 2005. – 180 p. – Available from: http://www.redbooks.ibm.com/redbooks/pdfs/sg246649.pdf. – 16.10.2019.
5. Радченко, Г. И. Распределенные вычислительные системы / Г. И. Радченко. – Челябинск : Фотохудожник, 2012. – 184 с.
6. King, J. L. Centralized versus decentralized computing: organizational considerations and management options / J. L. King // ACM Computing Surveys. – 1983. – Vol. 15, Issue 4. – P. 319–349.
7. Foster, I. The Anatomy of the Grid: Enabling Scalable Virtual Organizations / I. Foster, C. Kesselman, S. Tuecke // International Journal of Supercomputer Applications and High Performance Computing. – 2001. – Vol. 15, № 3. – P. 200–222.
8. Advances in heterogeneous network computing / P. Gray, A. Krantz, S. Olesen, V. Sunderam // Lecture Notes in Computer Science. – Springer Berlin Heidelberg, 1998. – Vol. 1497. – P. 83–92.
9. From the I-WAY to the National Technology Grid / R. Stevens, P. Woodward, T. DeFanti, C. Catlett // Communications of the ACM. – 1997. – Vol. 40, Issue 11. – P. 50–60.
10. US4405829 A.Cryptographic communications system and method / R. L. Rivest, A. Shamir, L. M. Adleman (USA); patent holder Massachusetts Institute Of Technology. – US 05/860,586; claimed 14.12.1977; published 20.09.1983. – U. S. Patient 4,405,829. – 20 p.
11. Software Infrastructure for the I-WAY High Performance Distributed Computing Experiment / I. Foster, J. Geisler, W. Nickless, W. Smith, S. Tuecke // Proc. 5th IEEE Symposium on High Performance Distributed Computing, 6–9 Aug. 1996. – Syracuse, NY, USA, 1996. – P. 562–571.
12. Foster, I. Globus Toolkit Version 4: Software for Service-Oriented Systems / I. Foster // IFIP International Conference on Network and Parallel Computing, Nov. 30 – Dec. 3 2005. – Beijing, China, 2005. – Vol. 3779. – P. 2–13.

13. Grimshaw, A. The Legion Vision of a Worldwide Virtual Computer / A. Grimshaw, W. Wulf // Communications of the ACM. – 1997. – Vol. 40 (1). – P. 39–45.
14. Henning, M. The Rise and Fall of CORBA / M. Henning // ACM Queue. – 2006. – Vol. 4, Num. 5. – P. 28–34.
15. Java Remote Method Invocation (Java RMI) [Electronic Resource]. – Available from: http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/. – 16.10.2019.
16. Когаловский, М. Р. Перспективные технологии информационных систем / М. Р. Когаловский. – М. : ДМК Пресс; Компания АйТи, 2003. – 288 с.
17. Тонкий клиент [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Тонкий клиент. – 16.10.2019.
18. Елманова, Н. Серверы приложений ведущих производителей [Электронный ресурс] / Н. Елманова // КомпьютерПресс. – 2003. – №10. – Режим доступа: http://compress.ru/article.aspx?id=12086. – 16.10.2019.
19. Application Servers (appservers) [Electronic Resource]. – Available from: http://www.bestpricecomputers.co.uk/glossary/application-server.htm. – 16.10.2019.
20. Серверы приложений [Электронный ресурс]. – Режим доступа: http://www.4stud.info/networking/application-server.html. – 16.10.2019.
21. Java EE Compatibility [Electronic Resource]. – Available from: http://www.oracle.com/technetwork/java/javaee/overview/compatibility-jsp-136984.html. – 16.10.2019.
22. Троелсен, Э. Язык программирования C# 5.0 и платформа .NET 4.5 / Э. Троелсен. – 6-е изд. – М.: Вильямс, 2013. – 1312 с.
23. Колесов, А. Рынок серверов приложений в исследовании Gartner [Электронный ресурс] / А. Колесов // PC Week/RE. – 2010. – № 11 (713). – Режим доступа: http://www.pcweek.ru/idea/article/detail.php?ID=121123. – 16.10.2019.
24. Удаленный вызов процедур [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Удаленный вызов процедур. – 16.10.2019.
25. Безверхов, М. Архив статей «Что такое технология COM?» [Электронный ресурс] / М. Безверхов. – Режим доступа: http://www.developing.ru/com/. – 16.10.2019.
26. Семихатов, С. Краткое введение в технологию Enterprise JavaBeans [Электронный ресурс] / С. Семихатов. – Режим доступа: http://www.javable.com/columns/serv_side/workshop/05/. – 16.10.2019.
27. Селютин, А. В. Распределённые системы объектов. DCOM [Электронный ресурс] / А. В. Селютин. – Режим доступа: http://masters.donntu.org/2008/fvti/selyutin/library/distr_sys.htm. – 16.10.2019.

28. COM+ (Component Services) [Electronic Resource]. – Available from: https://msdn.microsoft.com/en-us/library/windows/desktop/ms685978%28v=vs.85%29.aspx. – 16.10.2019.
29. Марков, Е. Технология COM+ (Microsoft Transaction Server) [Электронный ресурс] / Е. Марков. – Режим доступа: http://citforum.ru/operating_systems/windows/complus/. – 16.10.2019.
30. Печерский, А. Язык XML – практическое введение [Электронный ресурс] / А. Печерский. – Режим доступа: http://www.codenet.ru/webmast/xml/.– 16.10.2019.
31. OASIS UDDI Specifications TC - Committee Specifications [Electronic Resource]. – Available from: https://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm. – 16.10.2019.
32. Foster, I. The Grid. Blueprint for a new computing infrastructure / I. Foster, C. Kesselman. – San Francisco: Morgan Kaufman, 1999. – 677 p.
33. Stockinger, H. Defining the Grid : A Snapshot on the Current View / H. Stockinger // The Journal of Super-computing. – 2007. – № 42 (1). – P. 3–17.
34. Foster, I. Service-Oriented Science / I. Foster // Science. – 2005. – Vol. 308, № 5723. – P. 814–817.
35. Черняк, Л. Web-сервисы, grid-сервисы и другие / Л. Черняк // Открытые системы. СУБД. – 2004. – № 12. – С. 20–27.
36. Enhanced resource management capabilities using standardized job management and data access interfaces within UNICORE Grids / M. S. Memon et al. // 13th International Conference on Parallel and Distributed Systems, Dec. 5–7 2007. – Hsinchu, Taiwan, 2007. – Vol. 2. – P. 1–6.
37. Mohamed, A. A history of cloud computing [Electronic resource] / A. Mohamed. – Available from: http://www.computerweekly.com/feature/A-history-of-cloud-computing. – 13.10.2019.
38. Облачные вычисления, краткий обзор или статья для начальника [Электронный ресурс]. – Режим доступа: http://habrahabr.ru/post/111274/.– 14.10.2019.
39. A break in the clouds: towards a cloud definition / L. M. Vaquero et al. // ACM SIGCOMM Computer Communication Review. – 2009. – Vol. 39. – P. 50–55.
40. Mell, P. The NIST Definition of Cloud Computing [Electronic Resource] / P. Mell, T. Grance // NIST Special Publication 800-145. – 2011. – Available from: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. – 15.10.2019.

ЗМІСТ

Навчальне видання

Абрамова Вікторія Валеріївна
Абрамов Сергій Клавдійович

РОЗПОДІЛЕНІ СЕРВІСНІ СИСТЕМИ

Редактор Т. О. Іващенко

Зв. план, 2020
Підписано до видання 5.11.2020
Ум. друк. арк. 6,1. Обл.-вид. арк. 6,81. Електронний ресурс

Видавець і виготовлювач
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»
61070, Харків-70, вул. Чкалова, 17
http://www.khai.edu
Видавничий центр «ХАІ»
61070, Харків-70, вул. Чкалова, 17
izdat@khai.edu

Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготовлювачів і розповсюджувачів
видавничої продукції сер. ДК № 391 від 30.02.2001