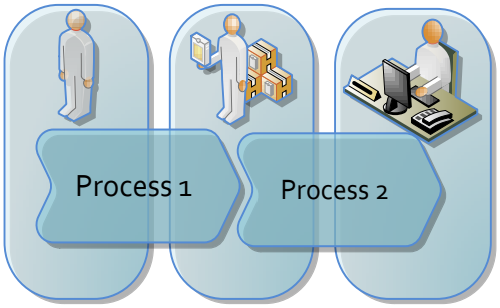
Литвинов О.А., Хандецький В.С.

# РОЗПОДІЛЕНА ОБРОБКА ІНФОРМАЦІЇ




**О.А. Литвинов, В.С. Хандецький**

# РОЗПОДІЛЕНА ОБРОБКА ІНФОРМАЦІЇ

**Монографія**

**ТОВ «Баланс-Клуб»**

**Дніпропетровськ**

**2013**

УДК 004.9
ББК 32.973.202
Л 64

*Друкується за рішенням вченої ради*
*Дніпропетровського національного університету ім. Олеся Гончара*
*(протокол №6 від 27.12.2012р )*

**Рецензенти:**
д-р фіз.-мат. наук, проф. В.І. Гаврилюк
(завідувач кафедри автоматики, телемеханіки і зв'язку Дніпропетровського національного університету залізничного транспорта імені академіка В. Лазаряна),
д-р фіз.-мат. наук, проф. О.В. Коваленко
(завідувач кафедри радіоелектроніки Дніпропетровського національного університету імені Олеся Гончара)

**Литвинов О.А., Хандецький В.С.**

Розподілена обробка інформації : [моногр.] / О.А. Литвинов.,
Л64 В.С. Хандецький − Д.: ТОВ «Баланс-Клуб», 2013.− 314 с.

ISBN 978-966-494-025-9

Монографію присвячено дослідженню основних класів розподілених обчислювальних систем: систем, що спрямовані на високопродуктивні обчислення та розподілених інформаційних систем. Особлива увага приділена аналізу підходів, моделей та технологій сучасних розподілених інформаційних систем та їх ролі у побудові інфраструктури бізнесу. Розглянуті тенденції розвитку розподілених обчислень. Монографія може бути корисна інженерам та аналітикам комп'ютерних систем, що спеціалізуються в галузі високопродуктивних обчислень, студентам інформаційних спеціальностей вищих навчальних закладів України.

УДК 004.9
ББК 32.973.202

ISBN 978-966-494-025-9

# ЗМІСТ

# ВСТУП

Важливими рисами сучасних підходів до управління підприємством, які базуються на моделі управління бізнес процесами, являються: повна формалізація активностей; всебічне моделювання заходів перед їх впровадженням; застосування механізмів моніторингу та контролю якості виробництва, які спрямовані на оптимізацію існуючих бізнес процесів. Такі заходи неможливі без повної інтеграції бізнесу з інформаційними технологіями, які створюють IT-інфраструктуру/контекст бізнесу. Вони спрямовані насамперед на забезпечення підтримки фаз моніторингу та контролю якості, створюючи фактологічне та аналітичне підґрунтя для реструктуризації та оптимізації бізнес процесів.

У разі такої інтеграції виникає ряд питань, без вирішення яких IT-інфраструктура стає тягарем для розвитку бізнесу: гнучкість - досягнення відповідної реакції системи на зміни бізнес - процесів, безпека, надійність, доступність, масштабованість, відкритість. Виконання цих умов неможливо без залучення сучасних підходів до проектування та IT-технологій (методів, підходів, засобів). Розвиток та ускладнення бізнес-процесів, високі вимоги до складності та швидкості обчислень, збереження великого об’єму даних, інтеграції з іншими системами призводять до виключної ролі розподілених обчислювальних систем, комплексів гетерогенних реальних та віртуальних обчислювальних машин, як основного компоненту IT-інфраструктури.

При цьому загальною стратегією розробників IT-інфраструктури є: створення нових систем з вже існуючих функціональних компонентів і поєднанням їх на базі стандартних протоколів; використання парадигм проектування та програмування, спрямованих на швидку адаптацію до змін; використання інтегрованих середовищ розробки та адміністрування; віртуалізація на різних системних рівнях; введення різноманітних високорівневих мов та різноманітних метафор для ясного описання та розуміння обчислювальної задачі та її рішення.

У даній роботі розглядаються основні класи розподілених обчислюва-

льних систем: системи, що спрямовані на високопродуктивні обчислення та розподілені інформаційні системи. Особлива увага приділена підходам, моделям та технологіям сучасних розподілених інформаційних систем, їх ролі у побудові інфраструктури сучасного бізнесу, без розуміння якої неможливо отримати повну картину сучасного стану та тенденцій їх розвитку.

В розділі 1 вводяться основні поняття та характеристики (критерії оцінювання) розподілених обчислювальних систем.

В розділі 2 розглядаються основні принципи побудови інформаційних моделей управління бізнес-процесами підприємств, особлива увага приділена моделі оптимізації продуктивності бізнесу і моделі оптимізації інфраструктури додатків.

В розділі 3 розглядаються кластери, Grid та хмарні системи. В частині, що пов'язана з кластерними обчисленнями поряд з високопродуктивними системами (нащадками кластеру Beowulf) розглянуто також відмовостійкі кластери, кластери балансування навантаження, віртуалізацію серверів, як невід’ємні механізми побудови сучасних розподілених систем .

Розділ 4 присвячений класу розподілених інформаційних систем, в якому розглядаються: сутність та проблеми обробки розподілених транзакцій; особливості обробки транзакцій в Microsoft SQL Server, в системі Windows.

В розділі 5 розглянуто структуру і характеристики нетранзакційних розподілені бази даних і принцип BASE, проаналізовано особливості підходу Scale – out для реляційних СУБД.

Розділ 6 присвячений інтеграції корпоративних додатків. В ньому розглянуто архітектури систем федеративних баз даних; системи, орієнтовані на виклики віддалених процедур; RPC – системи.

В розділі 7 розглянуто системну архітектуру розподілених систем: клієнт-серверну та однорангову моделі. Особлива увага приділена клієнт-серверній парадигмі – розглянуто основні класи систем: «віддалена оцінка», «код за запитом», агенто-орієнтовані системи.

В розділі 8 надається концепція сервісно-орієнтованої архітектури (далі СОА), принципи якої спрямовані на оптимальну підтримку бізнес-процесів, побудову гнучких розподілених гетерогенних програмних комплексів. Особлива увага сфокусована на зв'язок СОА з моделлю управління підприємства на основі бізнес-процесів. Проаналізовано технології веб-сервісів ASP.NET для побудови додатків. Розглянуті особливості, пов'язані як з побудовою класичних SOAP-, так і з REST-сервісів: використання механізмів кешування, відстрочене виконання, виконання розподілених транзакцій, синхронні й асинхронні http-оброблювачі.

В розділі 9 наведені практичні приклади застосування Web – сервісів з використанням ASP.NET. Зокрема розглянуто процедури ідентифікації користувача, створення рівню сервісу, створення тестеру клієнта, наведені завдання для самостійної підготовки.

В розділі 10 включено практичний матеріал, пов'язаний із застосуванням сокетів, як базового механізму мережевої взаємодії в сучасних інформаційних системах. Розглянуто процедури роботи з сокетами в .NET, практичні задачі щодо отримання списку адрес машини з використанням DNS, створення типового додатку з потоковим типом взаємодії, організації взаємодії з використанням протоколу UDP. Наведені завдання для самостійної підготовки.

# 1. Призначення та особливості розподілених систем

***Розподіленою обчислювальною системою (РОС)*** називають множину незалежних ресурсів, що можуть взаємодіяти між собою з метою спільного рішення обчислювальних завдань. У більш вузькому сенсі, РОС - це множина автономних машин-обчислювачів (вузлів), об'єднаних мережею з установленим спеціалізованим програмним забезпеченням, які складають цільне, погоджене середовище для вирішення обчислювальних задач. У ряді випадків розподілені системи дозволяють забезпечити більш ефективне виконання задач у порівнянні з багатопроцесорними системами, основними перевагами при цьому являються гнучкість організації системи та масштабованість ресурсів. Слід відзначити, що ***ключовим компонентом (хребтом)*** такої системи є відповідний шар програмного забезпечення (ПЗ), що надає можливість скоординованої роботи визначеної кількості гетерогенних ресурсів - фізичних обчислювачів та програмних додатків, які функціонують під керуванням операційних систем (ОС) (рис.1). Цей шар базується на ***стандартах*** щодо на представлення та обміну (representation and exchange) інформацією. Особливостями ресурсів є можливість незалежного використання, скоординована робота (collaboration) з метою вирішення обчислювальної задачі, доступність до кінцевого користувача. Об’єднання ресурсів повинно здійснюватись у вигляді єдиної, несуперечливої (coherent) системи. Важливою складовою вищеозначеного шару ПЗ РОС являється набір методів, підходів, засобів розподілення та управління вирішенням обчислювальних завдань (tasks), множина яких складає обчислювальну задачу (computing problem). Безумовно ПЗ РОС можливо розглядати на різних рівнях, виділити додаткові шари, класи компонентів залежно від їх специфіки. Але основу такого ПЗ складає ряд сервісів, які формують інтерфейс взаємодії компонентів різних програмних додатків та вирішують проблеми, пов’язані з відмінностями апаратних та програмних платформ.

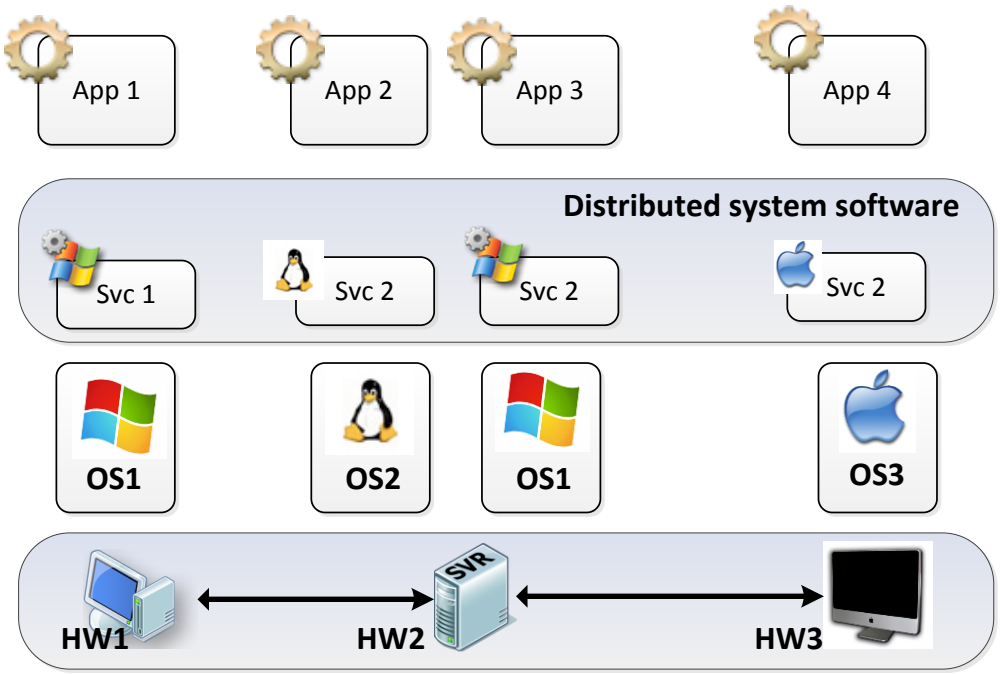



Рис. 1. Модель розподіленої системи.

Ключовими характеристиками розподілених систем являються: прозорість, відкритість, паралельність виконання завдань, масштабованість, стійкість до помилок [1]. Розглянемо ці характеристики.

***Прозорість (Transparency).*** Дана характеристика припускає приховування деталей реалізації, структури й функціонування системи. Щодо структурної складової системи - приховуються особливості представлення даних, наявність ряду різнорідних програмних і апаратних ресурсів, особливості їх конкурентного використання. Стосовно процесів − приховуються наявність паралельної обробки (concurrency), міграції (migration) або переміщення (reallocation) ресурсів. Міграція на відмінність від переміщення означає «живе» переміщення ресурсу або обчислювального процесу на інший вузол РОС без зупинки процесу користування (наприклад, live migration в системі Microsoft Hyper-V користувальницьких віртуальних машин з одного вузла кластера на інший для збалансування потужності).

Важливою функцією розподіленої системи є приховування деталей та навіть факту розподілення обчислень між багатьма апаратними та програм-

ними ресурсами, деталей координації їх роботи. Основні види прозорості, які забезпечуються розподіленими системами наведені у таблиці 1.

Типи прозорості

Таблиця 1

| Тип прозорості | Пояснення |
|---|---|
| Доступу (Access) | Приховує різноманіття в представленні даних та в питаннях доступу до ресурсів |
| Розміщення (Location) | Дозволяє забезпечити доступ до локальної та віддаленої інформації в уніфікованому вигляді незалежно від місця та часу взаємодії |
| Помилок (Failure) | Дозволяє автоматично виявити і виправити помилки, створюючи при цьому у користувача враження безпомилкової роботи системи |
| Реплікації (Replication) | Дозволяє невидимо дублювати програмне забезпечення і дані на ряді машин з метою підвищення продуктивності |
| Міграції (Migration) | Приховує переміщення ресурсу на інше місце |

***Відкритість (Openness).*** Відкритість системи для розвитку(evolution) та використання означає її незалежність від специфіки ресурсів, доступність для використання програмними додатками користувача. Відкритість досягається шляхом введення специфікації програмної взаємодії. При цьому слід відзначити такі характеристики, як портируємість (portability) і інтероперабельність (interoperability) компонентів системи. *Портируємість* - характеристика, що оцінює здатність переносу системи з однієї платформи (програмної або апаратної) на другу. Так програмний модуль називається портируємим у випадку, якщо його адаптація до нових умов (платформи) менша ніж ціна переробки даного продукту. *Інтероперабельність* - оцінює можливість взаємодії системи з іншими системами, незважаючи на можливі розходження, які пов'язані з їхньою реалізацією (мова програмування, середовище виконання). Обидві характеристики пов’язані зі спроможністю повторного використання продукту (reuse) у новому середовищі, або безпосередньо, або за новими

умовами, відмінними від тих, за якими він був створений.

***Паралельність обчислень (Concurrency).*** Паралельність виконання досягається шляхом паралельного розсилання запитів на виконання завдань на ряд зв'язаних обчислювачів.

***Масштабованість (Scalability).*** Якщо при наявності певної кількості машин обчислювальна потужність системи недостатня, то можна її розширити шляхом додавання нових апаратних та програмних ресурсів.

***Обробка помилок (Fault-tolerance).*** Робочі машини можуть приймати на себе роль резервних обчислювачів у випадку виходу з ладу інших машин (через збій апаратного або програмного забезпечення). У такому випадку говорять, що збої і помилки можуть бути виявлені й оброблені іншими машинами. Головною ідеєю являється приховування від користувача та обробка виключних ситуацій: прогнозування виключних ситуацій, відновлення роботи процесів та їх станів, розподілення навантаження на інші вузли.

До переваг використання розподілених систем слід віднести:

- оптимізацію використання ресурсів, яка досягається використанням незадіяних потужностей (процесори), ємностей накопичувачів (оперативної або дискової пам'яті), графічних можливостей (графічні процесори), периферійних пристроїв (принтери, плотери);
- підвищення швидкості доступу до даних за рахунок використання розподілених баз даних і широкомовних запитів/відповідей;
- підвищення інформаційної ємності системи за рахунок збільшення кількості вузлів мережі;
- підвищення швидкості обробки даних, що обумовлює збільшення загальних обчислювальних можливостей системи щодо окремих її вузлів;
- підвищення стійкості системи, яка досягається шляхом організації децентралізованої обробки.

Розподілені системи мають довгу історію розвитку (1966 рік), але пік розвитку може бути визначений широким поширенням мережі Інтернет, яке, в свою чергу, обумовлено появою потужних персональних та серверних

комп'ютерів, швидких локальних і глобальних мереж, відповідного програмного забезпечення.

# 2. Інформаційні моделі управління бізнес-процесами підприємств

## 2.1. Принципи побудови оптимізаційних моделей

Одним з інструментів для оцінки тенденцій розвитку інформаційних технологій, у тому числі і технологій розподілених обчислень, являється набір моделей оптимізації інфраструктури, запропонований фірмою Microsoft для визначення рівня розвитку інфраструктури підприємства, та шляхів її покращення. Інформаційні технології за відомим слоганом Microsoft служать бізнесу, основною рисою якого являється мінливість (agility). Згідно з моделлю управління бізнес процесами (business process management model), яка застосовується в більшості ведучих американських компаній, фази виконання та моніторингу бізнесу є тісно пов'язаними з інформаційними технологіями. А згідно з дослідженнями Гартнер [Gartner, 2010] ("2010 CIO Survey: A Time of Great IT Transition") організація IT повинна давати додатковий ефект, не вимагаючи при цьому додаткових витрат. Для реалізації зростаючих вимог бізнесу слід застосовувати нові методичні підходи, які дозволяють досягнути максимальних цілей бізнесу з мінімальними витратами на побудову та підтримку IT інфраструктури.

З метою визначення можливих напрямків розвитку IT інфраструктури підприємства Microsoft пропонує дві ключові моделі: інфраструктура продуктивності бізнесу (Business Productivity Infrastructure - BPIO) та основна модель інфраструктури (Core Infrastructure - Core IO). Ці моделі ілюструють стратегічну цінність та переваги, які отримує бізнес в результаті здійснення переходу від базового рівня оптимізації з хаосом технологій та необґрунтованістю витрат до динамічної інфраструктури, де цінність інфраструктури для бізнесу має ясне визначення і обґрунтування. Деякі автори оглядів та пропагандисти Microsoft використовують термін шляхова карта IT (IT

roadmap), яка дозволяє начальникам інформаційних відділів підприємств обрати правильний шлях до успіху.

Кожна з моделей має ключові можливості (key capabilities) та рівні навантаження (levels workloads), які дозволяють відобразити бізнес-фактори та пріоритети на технологічні рішення, що можуть формуються на базі функціональних можливостей платформи.

### 2.2. Модель базової оптимізації інфраструктури (CIO)

Розглянемо модель базової оптимізації інфраструктури (Core Infrastr - cture Optimization). Вона спрямована на побудову та оптимізацію базової інфраструктури підприємства, включаючи управління мережевими ресурсами, організацію безпечної мережі. В основу оптимізації покладено продукт Windows Server, який підтримує велику кількість мережевих служб. Для управління інфраструктурою застосовуються такі продукти, як Microsoft System Management Server, Microsoft Operations Manager. При цьому визначають наступні характеристики.

***Управління та віртуалізація центрів обробки даних*** **(*Datacenter Management and Virtualization*)** сфокусовані на кращих практиках управління та обслуговування серверів, пристроїв зберігання даних, мережевих компонентів, та інших апаратних і програмних ресурсів центру обробки даних. Вони спрямовані на забезпечення консолідації всіх інструментів управління та якісної звітності, оцінку гетерогенних компонентів системи, а також високої доступності сервісу. Для цього застосовуються технології кластеризації та балансування навантаження.

***Розгортання та управління пристроями*** **(*Device Deployment and Management*)** сфокусоване на управління апаратно-програмним забезпеченням та політикою користувальницького доступу. Поширення пристроїв IT диктує вимоги до введення стратегії блокування, шифрування та забезпечення безпеки пристрою, узгодження конфігурацій.

***Служби ідентифікації та безпеки* (*Identity and Security Services*)** описують кращі практики організації управління та захисту, ідентифікації користувачів і авторизації бізнес-ресурсів і додатків. Це також допомагає керувати доступом для корпоративних мобільних користувачів, клієнтів і партнерів за межами брандмауера. Ця можливість охоплює методи і стратегії для захисту різної інформації, яка знаходиться всередині організації на серверах, клієнтських системах чи мережевих ресурсах.

***Процеси та узгодження IT* (*IT Process and Compliance*)** надають основу для організації планування, розробки, розгортання та підтримки IT-послуг і рішень, допомагаючи забезпечити високу надійність і доступність. Це також допомагає організаціям узгоджувати елементи управління IT з бізнесом та визначити політику і надійність для IT звітності.

Розглянемо особливості організації рівнів навантаження (levels workloads) з метою оптимізації інфраструктури IT.

Можна виділити наступні недоліки базової інфраструктури: ручне управління, локалізовані процеси, слабкий центральний контроль, відсутність або невиконання політик безпеки, резервного копіювання, розгортання сервісу та інших загальних IT стандартів. Існує загальний недолік знань про деталі інфраструктури, тому неможливо визначати тактику поліпшення сервісу. Через брак інструментів та ресурсів немає моніторингу стану додатків. Відсутній механізм розповсюдження накопичених знань через середовище IT. Користувачі базових інфраструктур визнають, що їх середовище надзвичайно складно контролювати, мають важку реалізацію клієнтської сторони, високі витрати на управління сервером, слабкий захист системи, дуже малий вплив на можливість покрашення бізнес процесу за допомогою IT. Взагалі будь-яке внесення змін в програмне забезпечення пов'язано з величезними витратами людських та фінансових ресурсів.

Перехід від базової до стандартизованої інфраструктури дозволяє значно скоротити витрати за рахунок:

- розробки стандартів, політики та управління згідно визначеної стратегії;

- зниження ризиків безпеки шляхом розробки «оборони в глибину» - багаторівневого підходу до безпеки: сервер – робоча станція – додаток;
- автоматизації багатьох ручних і трудомістких завдань;
- впровадження кращих практик управління сервісами (напр., ITIL - http://www.itil-officialsite.com/ і т.д.);
- спроби зробити IT стратегічним активом (asset).

Розглянемо наступні характеристики і визначення IT інфраструктури.

***Стандартизована інфраструктура*** запроваджує управління шляхом використання стандартів і політик керування робочими станціями та серверами, способів включення машин в мережу, розгортання служби Active Directory для керування ресурсами, застосування політики безпеки та контроль доступу. Користувачі стандартизованої інфраструктури працюють в рамках базових стандартів, але деякі політики, як і раніше, досить реактивні, тобто спрямовані на вирішення (реакцію) проблеми, а не на запобігання її виникнення. Внесення змін в програмне забезпечення, розгортання сервісу та обслуговування робочих станцій пов'язане із досить великими витратами людських та фінансових ресурсів. Тому на цьому рівні доцільне проведення інвентаризації апаратної та програмної складових, управління ліцензіями. Заходи зовнішньої безпеки, які розроблені за принципом «оборони в глибину» покращилася, але внутрішня безпека піддається ризикам.

При переході від стандартизованого до раціоналізованого стану, користувачі отримають значний контроль над інфраструктурою, визначені проактивні політики та процеси, починають їх підготовку до спектру можливих ризиків. Важливу роль грає при цьому впровадження концепції управління службами. Технологія також починає грати набагато більшу роль, стає важливою складовою, цінним активом бізнесу.

***Раціоналізована інфраструктура (Rationalized).*** Її метою є: значне зниження затрат, що пов'язані з управлінням робочих станцій і серверів; спрямованість процесів і політик на підтримку і розширення бізнесу. Безпека дуже активна і швидко реагує на загрози. Використання технології ZIT (Zero

Touch Deployment - http://technet.microsoft.com/en-us/library/dd919178 (v=ws.10).aspx) мінімізує витрати, час розгортання і технічні складнощі. Програмне забезпечення станцій класифіковане та зведене до певної кількості варіантів, управління відбувається за мінімальну кількість рухів. Вони мають прозору інвентаризацію апаратного та програмного забезпечення і тому здійснюється придбання тільки чітко визначених ліцензій та комп'ютерів. Безпека є надзвичайно активною (pro-active) з суворою політикою та контролем від настільного комп'ютера до сервера.

Користувачі виграють на рівні бізнесу, рухаючись від цього раціоналізованого стану до динамічного стану. Переваги від впровадження нових або альтернативних технологій, що беруть на себе управління бізнес - ризиками, набагато переважають додаткові витрати. Служба управління здійснюється за допомогою декількох сервісів з організацією заходів щодо більш широкого впровадження ІТ. Користувачі, які розглядають значення динамічного стану, зазвичай шукають моделі ІТ інфраструктури для забезпечення переваг їх бізнесу у конкурентній боротьбі.

***Динамічна інфраструктура.*** Користувачі з динамічною інфраструктурою в повній мірі усвідомлюють те стратегічне значення, яке може забезпечити ІТ інфраструктура, допомагаючи вести свій бізнес ефективно і залишатися попереду конкурентів. Витрати повністю контролюються, інтеграція між користувачами та даними, настільними комп'ютерами і серверами, співпраця між користувачами та відділами є широко поширеною, користувачі мобільних пристроїв мають сервіси та можливості на рівні звичайних операторів. Процеси повністю автоматизовані, часто вбудовані в саму технологію, що дозволяє ІТ швидко приладжуватися до змін бізнесу. Додаткові інвестиції в технології дають конкретні, швидкі, відчутні переваги для бізнесу. Використання власного резервування програмного забезпечення і формування карантинних систем (quarantine-like systems) для забезпечення виправлення помилок та дотримання встановлених політик безпеки дозволяє динамічно організовувати процеси, які підвищують надійність, знижують витрати і під-

вищують рівень обслуговування. Користувачі виграють від збільшення частки їх інфраструктури, що є динамічним, і забезпечує підвищений рівень сервісу, конкурентні переваги і більшу кількість бізнес-задач. Управління службами реалізоване для всіх критично важливих сервісів згідно угоди про рівень обслуговування (service level agreements) і встановлених операційних відгуків.

### 2.3. Модель оптимізації продуктивності бізнесу (BPIO)

Вона визначає принципи роботи компанії, які дозволяють підвищувати її продуктивність за рахунок більш тісної взаємодії між співробітниками, ефективного управління та аналізу розподілених даних. Технологічною платформою цієї моделі є продукти Microsoft Office, Microsoft Exchange Server, Microsoft SharePoint, Microsoft Dynamics CRM, Office Project Server, Office Performance Point. Ключовими можливостями є: співпраця, обмін повідомленнями, уніфікація зв'язку, управління контентом. Розглянемо складові цієї моделі.

***Співпраця (Collaboration)*** описує шляхи використання робочого простору (workspaces), засобів пошуку та порталів для забезпечення продуктивного середовища спільної роботи, яке відповідає конкретним вимогам бізнесу с застосуванням IT. При цьому пропонуються соціально-обчислювальні можливості, надаються стратегії для індексування та пошуку даних, інтеграції порталів з line-of-business додатками (LOB) і федеративних відносин з партнерами та клієнтами. Ця складова фокусується також на наданні користувачам якісного інформаційного (information-rich) середовища, розгорнутого на базі локальних та хмарних сервісів, засобів ефективного планування та управління з метою кращого узгодження з стратегіями та цілями бізнесу. ***Обмін повідомленнями (Messaging)*** – основний компонент внутрішніх і зовнішніх корпоративних комунікацій. Процедура сконцентрована на забезпеченні доступу до електронної пошти, розкладу, голосового спілкування, контактів різноманітним клієнтам та пристроям. Для цього розглядаються полі-

тики управління архівуванням, доступом до ресурсу, включаючи захист від шкідливих програм і забезпечення безпечної взаємодії.

***Уніфікація зв'язку (Unified Communication)*** забезпечує основу для спрощення порядку спільної роботи. Базується на застосуванні ряду інструментів для забезпечення конференцій, голосового спілкування та інш.

***Управління змістом/контентом підприємства (Enterprise Content Management)*** забезпечує основу для організації управління змістом (різноманітною інформацією, що використовується в роботі підприємства), який включає в себе різні типи даних та підходів до їх збереження, форматування, перетворення тощо. Визначає кращі процедури для управління повним життєвим циклом інформації - від виникнення, через зберігання та повторне використання, до видалення.

Рис.2 ілюструє перехід від базового рівня підприємства до динамічного.

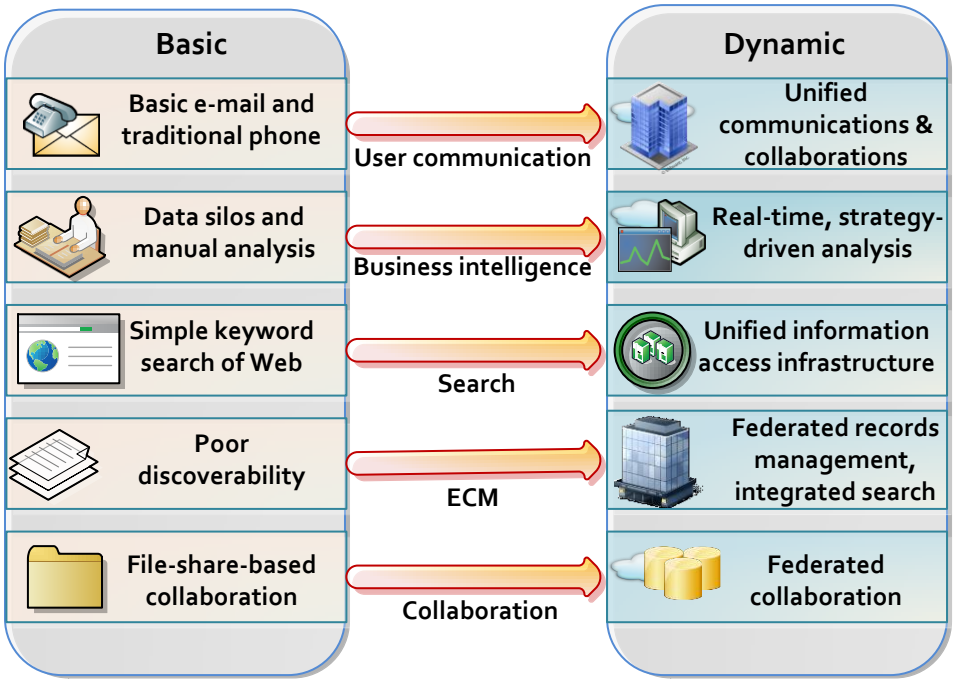



Рис.2. Перехід від базового рівня підприємства до динамічного.

Так поступово здійснюється перехід від застосування телефону та звичайного e-mail до інтероперабільної платформи бізнес-комунікацій з застосуванням ряду додатків (Microsoft Outlook, Microsoft Exchange); ручний аналіз неструктурованих даних змінюється на автоматичний аналіз, який відбува-

ється в реальному часі, та спрямований на стратегію підприємства; складний пошук необхідних документів змінюється інфраструктурою доступу до уніфікованої інформації; зберігання контенту у файлах та труднощі з пошуком та аналізом контенту – системою управління узагальненими документами та записами з інтегрованою підтримкою пошуку; взаємодія на базі розподілу файлів з застосуванням спеціалізованого середовища - на федеративну взаємодію (узагальнену взаємодію на базі узгоджень) за межами брандмауера з застосуванням мобільних клієнтів тощо.

### 2.4. Модель оптимізації інфраструктури додатків (APIO).

Ця модель сфокусована на побудову інноваційних рішень, розширення стандартної функціональності. Модель базується на продуктах Microsoft SQL Server, Microsoft Visual Studio, Microsoft BizTalk Server. Основними ключовими можливостями є: надання платформ для бізнес-аналітики, баз даних та бізнес-додатків, можливостей поширення існуючої системи за рахунок розробки користувальницьких сервісів. Розглянемо це більш докладно.

***Платформа для бізнес-аналітики (BI and Analytics Platform)*** надає користувачам можливість створення та ведення аналітики в масштабі всієї організації, використовуючи такі інструменти як Excel і SharePoint. Платформа націлена на управління життєвим циклом від доступу до даних, управління з використанням панелей самообслуговування (self-service dashboards), візуалізації даних, зручній звітності. Використання сховищ даних і вітрин даних (data warehouses and data marts), у тому числі для великих проектів дозволяють запроваджувати механізми виявлення прихованих залежностей, підтримки рішень.

***Платформа для баз даних і бізнес-додатків (LOB)*** дозволяє управляти базою даних з використанням інструментів, які забезпечують наочність та дозволяють більш ефективно управляти масштабуванням, балансуванням навантаження процесора і використання дискового простору, автоматизацією виконання завдань з використанням гнучких опцій конфігурації. Ця можли-

вість дозволяє встановити стратегії для підвищення продуктивності і масштабування обробки транзакцій, використання єдиної консолі для управління серверами, додатками і даними LOB-систем. Передбачено надання підтримки для розгортання, хостингу і управління додатками в різних операційних середовищах.

***Можливості поширення існуючої системи шляхом розробки користувальницьких додатків (Custom Development capabilities)*** спрямовані на покращення дизайну існуючих додатків або налаштування і/або інтеграції з іншими користувальницькими бізнес-додатками. До цього також відносяться: автоматизація процесів у рамках організації, організація центрів обробки даних і хмарних сервісів, інтеграція додатків і даних, використовуючи платформу розробки, яка дозволяє повторне використання різноманітних компонентів та архітектур.

Згідно APIO існують 4 рівня зрілості організації (Organization's maturity) (рис. 3).

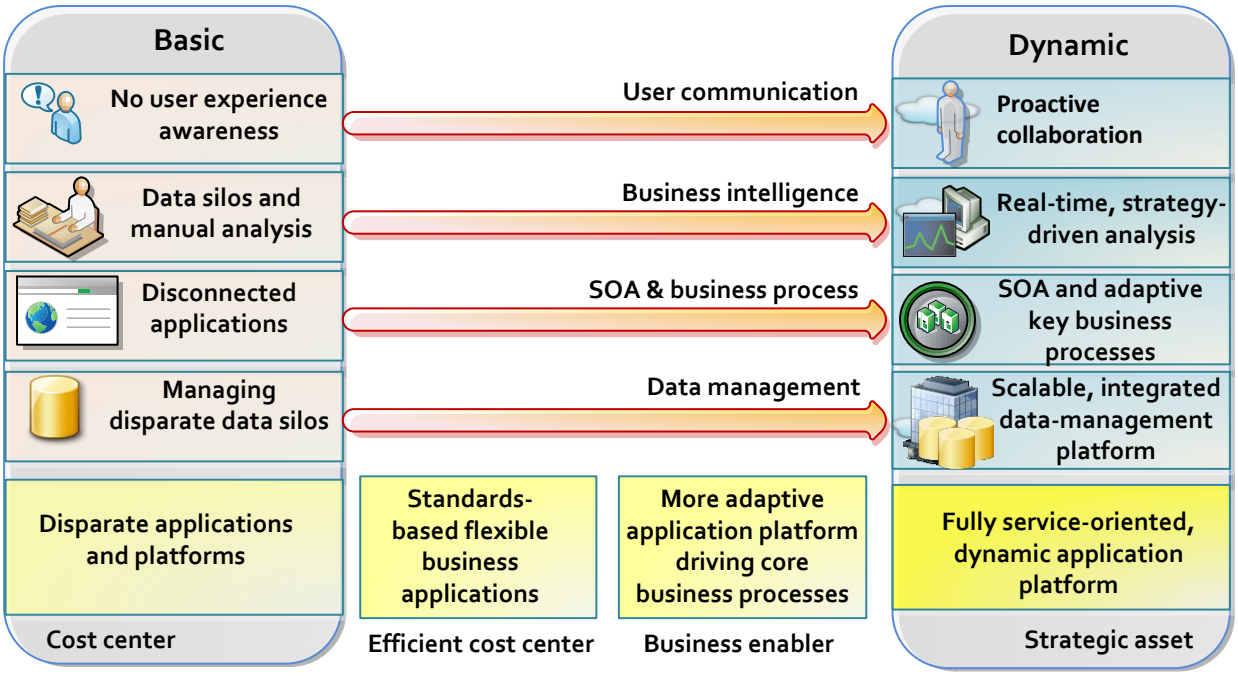



Рис. 3. Модель оптимізації інфраструктури додатків (APIO)

**Basic.** В організації існує безліч додатків (можливо на різних платформах) з невеликим ступенем інтеграції й повторного використання. Немає погоджених стандартів для інфраструктури й механізмів розробки (infrastructure or development techniques); немає цілісного архітектурного рішення.

**Standardized.** В організації робляться кроки щодо підвищення ефективності: з'являється архітектурний образ і робляться спроби для забезпечення повторного використання модулів/додатків.

**Advanced.** Програмне забезпечення для організації забезпечує підтримку бізнесу (business enabler). Чітка архітектура, велика кільеість сервісів і високий рівень повторного використання. Усі бізнес-процеси автоматизовані і можуть бути налагоджені. Використовується централізована платформа інтеграції.

**Dynamic**. Програмне забезпечення розглядається як стратегічна цінність організації. Зріла сервісна архітектура. Ясні процеси динамічних змін сервісів. Можливість швидкої адаптації до змін бізнес-вимог, швидкої інтеграції з новими бізнес партнерами.

## 3. Основні класи розподілених обчислювальних систем

На сьогодні існує багата кількість різноманітних технологій розподілених обчислень, доступу до даних, обміну файлами й повідомленнями. Але, незважаючи на разноманіття, виділяють три основних класи таких систем [Tanenbaum, 2006]: системи, спрямовані на високопродуктивні обчислення (high performance computing systems), розподілені інформаційні системи (distributed information systems), спеціалізовані розподілені системи (distr-?buted pervasive systems).

### 3.1. Системи, спрямовані на високопродуктивні обчислення

Такі системи в свою чергу підрозділяються на два основних підкласи: - **кластерні системи**, базове обладнання яких складається з сукупності однорангових робочих станцій (чи ПК) під управлінням однакової операційної

системи та зв’язаних високошвидкісною локальною мережею; **системи (grid, cloud), для обчислень на вимогу (on-demand comupting)**, які часто будуються як федерації комп’ютерних систем, де кожна система може відноситись до різних доменів, а також відрізнятися від інших і апаратним обладнанням.

### 3.1.1 Кластерні системи. Обчислювальні кластери

Історія створення кластерів пов'язана з ранніми розробками в галузі комп'ютерних мереж. На початку 1970-х рр. групою розробників протоколу TCP/IP (1975) і лабораторією Xerox PARC були закріплені стандарти мережевої взаємодії. У 1983 р. були створені механізми, що дозволяють з легкістю користуватися розподілом завдань і файлів через мережу в SunOS (з операційною системою на основі BSD від компанії Sun Microsystems). Першим комерційним проектом кластеру став Arcnet, створений компанією Datapoint в 1977 р. Прибутковим він не став, і тому будівництво кластерів не розвивалося до 1984 р., коли компанія DEC побудувала свій VAXcluster на основі операційної системи VAX/VMS. ARCNet і VAXcluster були розраховані не тільки на спільні обчислення, але й спільне використання файлової системи і периферії з урахуванням збереження цілісності і однозначності даних. VAXcluster (званий тепер VMSCluster) - є невід'ємною компонентою операційної системи HP OpenVMS.

Історія створення кластерів із звичайних персональних комп'ютерів багато в чому зобов'язана проекту PVM (Parallel Virtual Machine). У 1989 р. це ПЗ для об'єднання комп'ютерів у віртуальний суперкомп'ютер відкрило можливість миттєвого створення кластерів. Проект зробив можливим використовувати мережу гетерогенних машин під управлінням ОС Unix або Windows, як єдиний розподілений паралельний процесор. Створення кластерів на основі дешевих персональних комп'ютерів, об'єднаних мережею передачі даних, продовжилося в 1993 р. силами Американського аерокосмічного агентства (NASA). В 1994 р. групою Стерлінга і Беккера (Thomas Sterling, Donald

Becker) був розроблений кластер Beowulf, який складався із 16 комп'ютерів DX4 зв'язаних за допомогою мережі Ethernet. Ім’я Beowulf було запропоновано Стерлінгом та походить від головного героя старої англійської поеми Beowulf, який мав силу 30 чоловіків. Як правило, кластери Beowulf (рис.4) працюють на Unix-подібних операційних системах, таких як BSD, Linux чи Solaris, зазвичай побудовані за допомогою програмного забезпечення з відкритим кодом. Широко використовуються бібліотеки паралельних обчислень Message Passing Interface (MPI) та Parallel Virtual Machine (PVM). Обидві дозволяють програмісту розділяти задачу поміж групи мережних комп’ютерів та збирати результати обчислень.

Прикладами програмного забезпечення, яке реалізовує стандарт MPI є набір бібліотек OpenMPI (сайт проекту - http://www.open-mpi.org/) та MPI Chameleon від Argonne National Laboratory - MPICH (сайт проекту - http://www.mcs.anl.gov/research/ projects/mpich2/). Операційні системи Microsoft HPC Server 2008 та Windows Compute Cluster Server 2003 також включають реалізацію стандарту MPI, яка базується на продукті MPICH2 , але запроваджує додаткові можливості, такі як: безпека з застосуванням сервісів Active Directory, висока продуктивність обчислень на системах Windows (додаткова інформація доступна на сайті http://msdn.microsoft.com/en-us/library/ff976568.aspx).

В цілому особливостями обчислювальних кластерів є наступні:

- базове устаткування складається з ряду однорідних обчислювачів - робочих станцій або PCs, пов'язаних високошвидкісною локальною мережею ( як правило Ethernet);
- вузли розподіляються на три класи: точка доступу, мастер, виконавець,- мастер відповідає за розподіл завдань, моніторинг робіт, балансування навантаження;
- серверний вузол з'єднаний з зовнішньою мережею використовує дві карти Ethernet ;

• головний вузол (master) організує виконання програм та управлінням кластером, в той час, як комп'ютерні вузли, часто не мають потреби ні в чому, окрім стандартної операційної системи; у своїй роботі master використовує бібліотеки паралельних програм;

• моделлю програмування є SPMD (single process, multiple data).

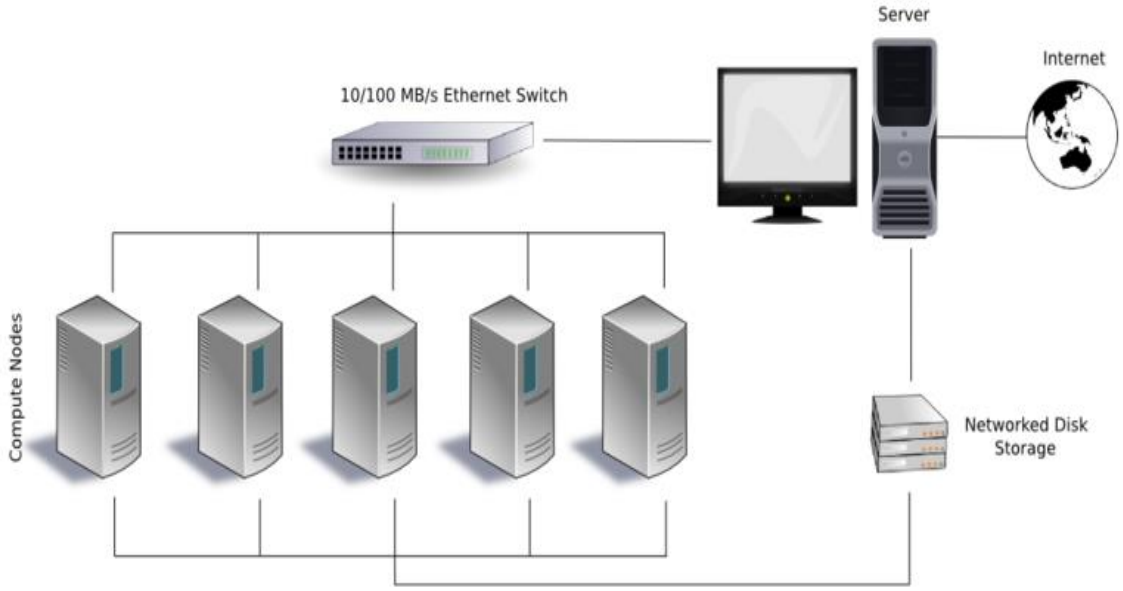




Рис.4. Кластер Beowulf.

Крім обчислювальних кластерів слід відзначити також два важливих різновиди кластерів - кластери високої доступності (high availability, failover clusters) та кластери розподілу навантаження (load balancing), які широко застосовуються для побудови обчислювальних комплексів та отримали значний розвиток останнім часом.

### 3.1.2. Кластери високої доступності

Призначенням відмовостійкого кластера високої доступності (далі КВД) є підвищення надійності і доступності критичних додатків. Як приклад можна привести бази даних, веб-сервери. При використанні технологій відмовостійкої кластеризації існує гарантія, що вихід з ладу одного фізичного сервера не приведе до відмови критичного додатку: додаток буде миттєво, або ж з невеликою затримкою (до декількох хвилин) перезавантажений на іншому сервері.

Відмовостійкий кластер складається як мінімум із двох серверів, які називаються вузлами кластера (nodes). Вузли кластеру періодично обмінюються між собою службовими пакетами інформації, іменованими heartbeat. Якщо один з вузлів виходить із ладу - він перестає відсилати пакети heartbeat, і кластер відзначає вузол як "збійний". При цьому, якщо на збійному вузлі були запущені критичні сервіси, то відбудеться їх автоматичний перезапуск на одному із працюючих вузлів. Всі вузли кластера використовують загальну систему зберігання даних, на якій зберігаються дані, необхідні для роботи самого кластера (так званий "кворум"), і дані, необхідні для функціонування критичних сервісів (файли БД, віртуальних машин, і т.д.). Хоча фізично кластер складається з декількох фізичних серверів - у мережі він представляється як один сервер, що має своє власне мережне ім'я й IP-адресу.

Визначають три основних класу відмовостійких кластерів :

• *з холодним резервом або активний / пасивний.* Пасивний вузол включається в роботу, коли відмовляє активний. Використовується зв’язка пакетів Linux DRBD (Distributed Replicated Block Device) що забезпечує постійну синхронізацію, та HeartBeat, що відповідає за виявлення збоїв у системі.

• *з гарячим резервом або активний / активний.* Всі вузли виконують власні запити, у разі відмови одного з них навантаження перерозподіляється між рештою. Тобто функціонує кластер розподілу навантаження з підтримкою перерозподілу запитів при відмові, наприклад Microsoft Cluster Server.

• *з модульною надмірністю.* Всі вузли одночасно виконують один і той самий запит, з результатів береться будь-який. Необхідно гарантувати, що результати різних вузлів завжди будуть однакові (або відмінності гарантовано не вплинуть на подальшу роботу). Прикладом є RAID (Redundant Array of Independent Disks) .

Слід відзначити, що використання описаної технології приводить до значного подорожчання системи: по-перше - через необхідність використання двох й більше серверів замість одного, по-друге - через необхідність зовнішньої системи зберігання даних, по-третє – через необхідність використання додаткового ПЗ. Але важливість характеристики високої доступності для користувача важко переоцінити, тому постачальники послуг, як правило, йдуть на додаткові витрати на обладнання та здійснюють пошук варіантів скорочення цих витрат за рахунок впровадження нових технологій (наприклад, віртуалізації серверів).

### 3.1.3. Кластери розподілу навантаження (КРН)

Метою розподілу навантаження є, в значній мірі, підтримка масштабованості системи. Клієнти роблять запит на підключення, використовуючи ім'я віртуальної мережі, підключення здійснюється до одного з вузлів кластера з балансуванням мережного навантаження.

Типовий випадок щодо використання кластеризації з балансуванням мережного навантаження - це створення веб-ферм на основі продукту Microsoft Web Farm Framework, який дозволяє визначити «серверну ферму» зі значною кількістю серверів. Ці сервери будуть автоматично обновлятися, забезпечуватися та керуватися, при цьому Web Farm Framework забезпечить конфігурацію, визначену на первинному сервері, автоматично продублює розгорнуті на ньому додатки та інші програмні продукти по всім серверам у веб- фермі. Web Farm Framework також включає в себе інтеграцію балансування навантаження (рис.5). Web Farm Framework може об’єднуватися для балансування навантаження з HTTP. Коли веб-сервери веб-ферми отримують

сигнал оновлення, вони можуть бути автоматично виведені з обороту балансування навантаження, здійснити процедуру оновлення і потім знову додані до складу ферми. Web Farm Framework також може вибірково оновлювати машини – так що система постійно буде мати доступні сервери для керування навантаженням.

Поточна версія Web Farm Framework включає вбудовану підтримку щодо маршрутизації запитів ARR (Application Request Routing), підтримує автоматичне балансування навантаження HTTP запитів по багатьом машинам на веб-фермі. ARR являється компонентом IIS (Internet Information Service) і може бути визначеним як проксі-модуль, спрямований на маршрутизацію запитів на основі заголовків HTTP запитів (HTTP headers), аналіз змінних параметрів сервера (server variables) та реалізацію алгоритмів балансування навантаження. Web Farm Framework полегшує інтеграцію веб-ферми серверів з службою ARR. Приклад балансування навантаженням наведено на схемі рис.3.

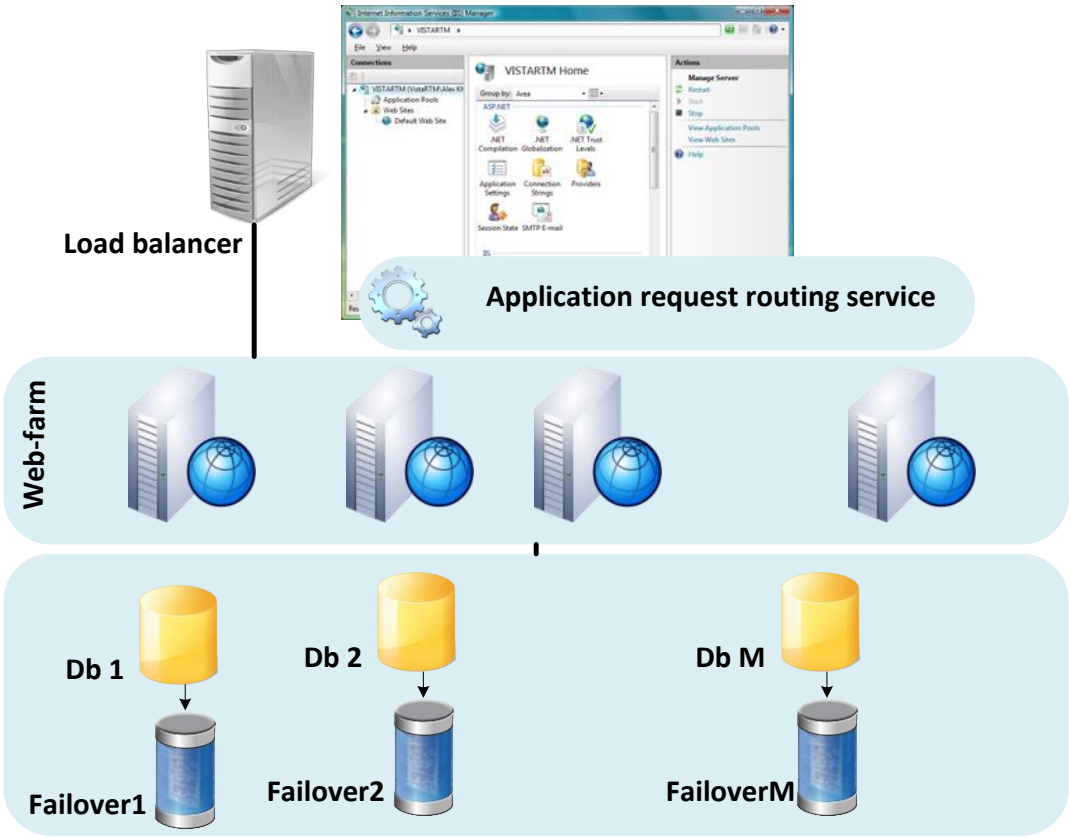



Рис. 5. Приклад балансування навантаження.

### 3.1.4. Кластеризація віртуальних серверів

Особливої уваги при розгляді тенденцій розвитку кластерних систем заслуговує створення кластерів на базі віртуальних серверів.

Віртуалізація серверів має на увазі запуск на одному фізичному сервері декількох віртуальних машин, які емулюють роботу реальних апаратних пристроїв, що функціонують під управлінням конкрктних ОС, виконують додатки. Для віртуалізації найбільщ часто використовуються продукти VmWare (ESX, Server, Workstation) і Microsoft (Hyper-V, Virtual Serer, Virtual PC). До переваг віртуалізації відносять наступне.

1. Зниження витрат на обладнання та обслуговування.

З впровадженням багатоядерних процесорів серверні технології вийшли на новий рівень розвитку. Зростання числа запитів, об'єму та складності обчислень, пікового навантаження, а також вимоги до адаптивності інформаційних систем до бізнесу диктують розділення систем на компоненти, які відповідають за виконання специфічних завдань (наприклад веб-сервера, сервера баз даних). Відображення цих компонентів на фізичні ресурси (серверні комп'ютери) залежить від пікового навантаження і призводить до простоїв обладнання, не виключаючи при цьому витрат на обслуговування з боку технічного персоналу та системних адміністраторів. Слід також відзначити, що процесорна стойка (rack) є споживачем великої кількості електроенергії та значним джерелом тепла, яке вимагає спеціальної системи охолодження, а впровадження компаніями певної кількості серверів пов'язане з побудовою спеціальних будівель. На рис.6. показані фотографії дата-центру Apple's Maiden (http://www.theregister.co.uk/2011/06/09/ apple_maiden_data_center/), 4-ох процесорних стійок U1, елементом якої є сервер ProServ M500 Small Business Rackmount 1U Server (http://www.servaris.com/servers_m500.php), який містить 2 сокети для процесорів Intel 5600 Six-Core Xeon та має 12 D?-DR3 DIMM слотів з можливістю розширення оперативної пам'яті до 192GB (96GB на 1 процесор).

Консолідація декількох віртуальних серверів на одному фізичному сервері дозволяє значно скоротити витрати на серверне обладнання та відповідно на його обслуговування. На одному фізичному сервері можуть одночасно функціонувати десятки віртуальних серверів, що дозволяє більш ефективно використати процесорний час і збільшити утилізацію ресурсів до 90%.

Рис.6. Приклад дата-центру з використанням серверних стійок типу U1.

2. Зниження витрат на програмне забезпечення.

Деякі виробники програмного забезпечення пропонують окремі схеми ліцензування спеціально для віртуальних середовищ. Так, наприклад, з придбанням однієї ліцензії на Microsoft Windows Server 2008 Enterprise, користувач

одержує право її використати на 1 фізичному сервері та на 4 віртуальних машинах (у межах одного фізичного сервера).

3.Зниження витрат на тестування нових платформ, середовищ [Kelby, 2009].

Шляхом використання механізмів зберігання та відновлення станів віртуальної машини (snapshot functionality) користувач отримує можливість здійснення будь-яких експериментів. В разі виникнення проблем збережений стан ВМ може бути переданий розробнику, який може безпосередньо бачити ситуацію, що склалася. Засоби WMI (Windows Management Interface) дозволяють здійснити тестування автоматично – завантажити ВМ, встановити останні оновлення, запустити на виконання тести.

3. Збільшення гнучкості інфраструктури.

Віртуалізація дозволяє програмному забезпеченню абстрагуватися від апаратної платформи. З'являється можливість міграції віртуальних машин між різними фізичними серверами. Без застосування віртуалізації при виході сервера з ладу доводиться переустановлювати ОС, відновлювати дані з резервних копій та інше – все це вимагає відповідних витрат.

4. Підвищення рівня відмовостійкості.

Віртуалізація надає можливість кластеризації сервера, незалежно від працюючого на ньому програмного забезпечення. Надається можливість кластеризації сервісів.

Необхідно також відзначити ***ряд особливостей, які слід враховувати при розгортанні віртуальних систем.*** До таких особливостей відноситься наступне.

*Забезпечення відповідальності.* На відміну від фізичних серверів, за кожний з яких відповідають певні фахівці, відповідальність за розгорнуті віртуальні системи часто буває неявною.

*Установка, відновлення й обслуговування.* Звичайні сервери обслуговуються й оновляються за установленим регламентом. Важливо забезпечити установку на віртуальні системи останніх відновлень й антивірусних баз.

*Видимість і сумісність.* Віртуальні сервери практично невидимі, тому мережна взаємодія між ними не завжди досить добре відслідковується. Наприклад може створитися така ситуація, коли одні конфігурації безпеки серверів можуть виявитися несумісними з іншими і захищені віртуальні сервери будуть взаємодіяти з відкритими в межах даного фізичного серверу. При цьому, щоб відстежити передачу мережних пакетів усередині цього фізичного сервера, необхідно впроваджувати додаткові заходи контролю.

*Безконтрольне поширення кількості віртуальних машин.* Відсутність належного розподілу відповідальності може спричинити зростання кількості віртуальних серверів, про деякі з яких після виконання ними своїх завдань просто забувають. Це створює додаткове навантаження на системні ресурси й приводить до появи небезпеки витоку конфіденційних даних. Щоб не допустити цього, необхідно застосовувати до віртуальної машини ті ж обґрунтування та впровадження корпоративних політик, що й до реальних серверів.

Microsoft Hyper-V являє собою рішення для віртуалізації серверів у корпоративних середовищах. Забезпечення суворого поділу між декількома ОС виконується шляхом створення віртуальних процесорів, пам'яті, таймерів і контролерів переривань. Операційні системи віртуальних машин використовують ці віртуальні ресурси так само, як використовували б їх фізичні аналоги. Віртуалізація здійснюється з застосуванням програмного забезпечення, яке називається монітором або гіпервізором віртуальних комп'ютерів (virtual machine manager, hypervisor) і відповідає за створення логічних розділів, управління виділенням пам'яті й ресурсів процесора для гостьових ОС, механізми віртуалізації введення/виведення й зв'язку між розділами, забезпечення виконання правил доступу до пам'яті; забезпечення виконання політики використання ресурсів центрального процесору (ЦП), програмний інтерфейс гіпервикликів. Hyper-V складається з одного *батьківського розділу* (host), що, по суті є віртуальним комп'ютером з прямим доступом до апаратних ресурсів. Всі інші віртуальні комп'ютери, відомі як *гостьові розділи* (guest), доступ до апаратних пристроїв здійснюють через батьківський розділ.

Hyper-V є програмним продуктом 64-х бітної версії Windows Server 2008, яка може працювати в режимі кластера.

Завдяки використанню загальної системи зберігання даних SAN (Storage Area Network) усіма вузлами кластера та застосування технології віртуалізації серверів під управлінням Hyper-V з'являється можливість побудови кластеру високої доступності. Створення такого кластеру не залежить від специфіки програмних платформ вузлів - операційної системи, сервісів, додатків. Справа у тому, що ряд додатків, наприклад, Microsoft Exchange Server або SQL Server самі по собі підтримують кластеризацію високої доступності, але більшість додатків кластеризацію не підтримують - саме в цьому разі і потрібно використовувати кластер віртуальних машин на базі Hyper-V. Так, у випадку виходу з ладу фізичного сервера з однією або декількома віртуальними машинами, відбувається автоматичний запуск цих віртуальних машин на іншому з існуючих робочих фізичних серверів. Важливою рисою цього підходу являється відсутність явного резервування фізичного серверу – всі сервери системи приймають участь в обробці даних, чим досягається балансування обчислювального навантаження. При цьому слід відзначити, що гостьова ОС може бути членом кластера з балансуванням навантаження тільки у разі використання в якості хостової ОС Windows Server 2008. Серед важливих характеристик ОС Windows Server 2008 R2 слід відзначити такі: підтримку довгої прив'язки (longer affinity) клієнта до вузла кластер ( за замовчуванням запити звичайно розподіляються по всіх вузлах) ; підтримку моніторингу додатків і сервісів; підтримку призначення вузлам цілого набору IP адрес.

Типовий приклад кластера на базі Hyper-V Server 2008 R2 показаний на рис.7. Node1 й Node2- вузли кластера, Server - хост для запуску віртуальної машини контролера домену DC.

На вузлах Node1 та Node2 установлений Hyper-V Server 2008 R2, на машині Server - Windows Server 2008 R2 з Hyper-V. Віртуальна машина DC, запущена на Server, використається: як контролер домену TEST.LOCAL та

як система зберігання даних. Для цього на ній установлено ПЗ Starwind iSCSi Target (http://www.starwindsoftware.com/). Безкоштовна версія дозволяє створювати iSCSi targets обсягом до 2Tb. Крім того до кластера з використанням iSCSi підключені два віртуальних дискових пристрої: обсягом 0,5Gb й 20Gb. Пристрій меншого обсягу використовується в якості кворумного ресурсу для роботи кластера, а більшого - для зберігання даних віртуальних машин.

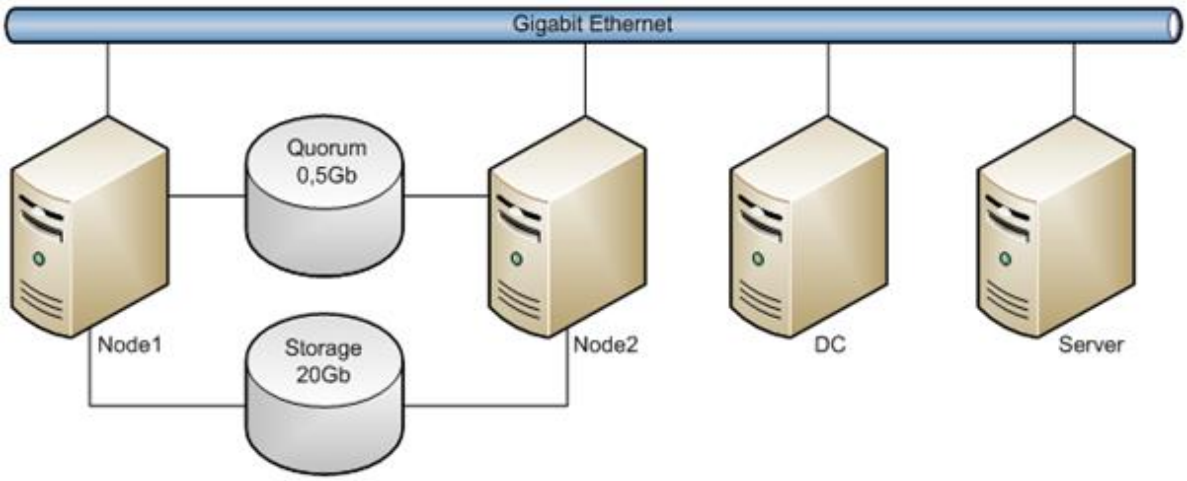



Рис. 7. Типовий приклад кластеру на базі Hyper-V Server 2008 R2.

Розглянемо роль кворуму більш детально. Важливою вимогою до кластеру високої доступності являється робота більшості його вузлів , якщо число функціонуючих вузлів кластера буде менш 50 відсотків, кластер вважається непрацездатним. Працездатність кластера визначається на базі аналізу кворуму. Основне призначення кворуму складається в виключенні проблем, пов'язаних з мережевою взаємодією вузлів кластера, так званою split-проблемою – у разі порушення мережі одна частина вузлів може мати взаємодію між собою, але не мати зв'язку з іншими вузлами, чим буде порушена цілісність кластеру. Для вирішення цієї проблеми використовується механізм кворуму, який базується на виділенні загальнодоступного логічного диску та встановленні спеціалізованого програмного забезпечення, яке реалізує алгоритм голосування (voting algorithm), що визначає працездатність кластеру.

Якщо число голосів, які подають працюючі вузли, менш визначеного порога – кластер припиняє роботу. При цьому робочі вузли продовжують роботу тільки у режимі отримання повідомлень.

Деталі роботи кворуму кластеру Windows Server 2008 R2 можна знайти на сайті http://technet.microsoft.com/en-us/library/cc730649(v=ws.10).aspx.

Сервери об'єднані мережею Gigabit Ethernet, Node1 й Node2 є членами домену TEST.LOCAL.

Відмовостійкий кластер Windows Server 2008 як правило розгортається в батьківському розділі (хост ВМ) Hyper-V. Це робиться з метою керування гостьовими віртуальними машинами (ВМ), їх переміщенням між вузлами кластера.

При профілактичних роботах, пов'язаних з фізичним комп'ютером, на якому працюють ВМ під керуванням Hyper-V, існує необхідність переміщення ВМ на інші вузли в кластері. У випадку збою фізичного комп'ютера, на якому працюють Hyper-V і ВМ, або серйозного зниження його продуктивності навантаження цього комп'ютера можуть бути переведені на інші вузли відмовостійкого кластера Windows, які автоматично приведуть їх у робочий стан.

У випадку збою віртуального комп'ютера його можна запустити знову на тому же сервері Hyper-V або перемістити на новий сервер Hyper-V. Оскільки відмовостійкий кластер Windows Server виявляє це, він автоматично робить поточні дії по відновленню, засновані на параметрах у властивостях ресурсів ВМ.

На рис.8 показана типова ситуація, коли віртуальна машина VM2 перебуває на комп'ютері розміщення Host A, потім VM2 переміщається на Host B. При цьому вузол, що володіє сховищем даних SAN LUN 2, де LUN (Logical Unit Number) - логічний пристрій у мережі сховищ даних (Storage Area Network), змінюється з Host A на Host B під час цього переміщення.

При роботі можна відмовитися від регулярного розміщення ВМ на деяких вузлах, тримаючи останні в резерві.

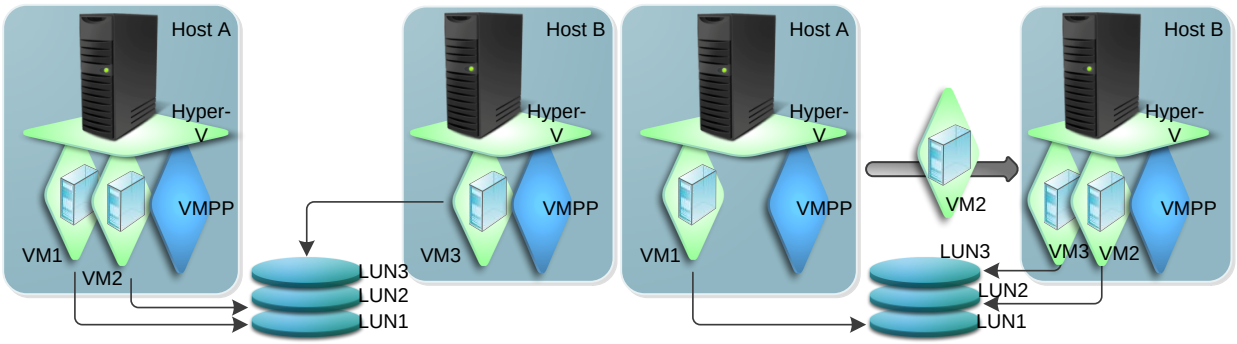



Рис. 8. Віртуальний хост і його сховище переносяться на новий вузол.

Можна розподілити ВМ по всіх вузлах, забезпечуючи наявність у кожного вузла достатньої додаткової ємності для успішного володіння віртуальними комп'ютерами і їхнім запуском при збоях будь-яких вузлів. Основні положення, пов'язані з кластеризацією мають наступний вигляд.

*Спостереження за працездатністю робочого навантаження*. У відмовостійкому кластері використовується монітор ресурсів, який робить виклики до бібліотеки ресурсу, пов'язаної з кластером. У кожному ресурсі є функція відстеження працездатності, яка перевіряє керований ресурсом додаток або службу з метою забезпечення коректної роботи додатку. Ці перевірки йменуються перевірками is Alive/looks Alive. Якщо при одному із цих викликів до ресурсу відбудеться збій, то відбудеться збій самого ресурсу. Залежно від того, як налаштовані властивості ресурсу, цей ресурс може спробувати повторно запустити відповідну службу чи додаток, або може бути переміщений на інший вузол відмовостійкого кластеру.

*Обслуговування* ВМ. Якщо необхідно змінити налаштування віртуального комп'ютера, відновити або змінити ОС або програмне забезпечення, робоче навантаження переміщується на інший вузол кластера з застосуванням механізмів швидкої або живої міграції (quick, live migration ).

*Збій ВМ або комп'ютера розміщення*. У випадку збою фізичного комп'ютера розміщення Hyper-V або гостьової системи ВМ інші вузли відмовостійкого кластера Windows виявлять, що член кластера більше не відповідає й

не бере участь у роботі кластера, після чого робочі вузли повернуть до роботи додаток або службу, що працювали на ВМ, яка вийшла з ладу.

*Досягнення високої доступності ВМ*. Настроювання ВМ на високу доступність полягають у використанні ролі майстра високої доступності в аплеті керуванні відмовостійкими кластерами. У віртуальних комп'ютерах Hyper-V є кілька ключових положень, які необхідно враховувати, коли вони управляються як високо доступні. Ці положення стосуються наступного.

Вузли відмовостійкого кластера. За кластеризацію комп'ютерів розміщення відповідає служба відмовостійкого кластера Windows Server 2008, яка встановлена на батьківській розділ системи Hyper-V. Це дозволяє гостьовим ВМ бути налаштованими як високодоступні віртуальні комп'ютери. ВМ, що налаштовані на високу доступність, будуть показані як ресурси в консолі керування відмовостійкого кластера.

Високодоступні сховища. Високодоступні віртуальні комп'ютери можуть бути налаштовані на використання віртуальних жорстких дисків (virtual hard disk - VHD), транзитних дисків і різницевих дисків. Щоб забезпечити переміщення віртуальних комп'ютерів між вузлами відмовостійкого кластера, є необхідним мати сховища з забезпеченням доступу до них з будь-якого вузла, що може розмістити віртуальний комп'ютер та який управляється службою відмовостійкого кластера. Транзитні диски варто додати до відмовостійкого кластера як дискові ресурси, на них повинні бути розміщені файли VHD.

Ресурс ВМ - це тип ресурсів відмовостійкого кластера, що представляє віртуальний комп'ютер. Коли ресурс ВМ починає роботу, Hyper-V створює дочірній розділ, а у віртуальному комп'ютері запускається відповідна ОС. При переході ресурсу ВМ у неробочий стан віртуальний комп'ютер видаляється з Hyper-V на вузлі, де він був розташований, а дочірній розділ видаляється з комп'ютера розміщення Hyper-V. Якщо ВМ відключена, зупинена або поміщена в збережений стан, цей ресурс буде знаходитись у неробочому режимі.

Ресурс налаштування ВМ. Являє собою тип ресурсу відмовостійкого кластера, який використовується для налаштування ВМ. Для кожної ВМ існує один ресурс її налаштування. Він указує шлях до файлу налаштування, що містить всю необхідну інформацію для додавання віртуального комп'ютера до комп'ютера розміщення Hyper-V. Оскільки налаштування управляється окремим ресурсом, його можна змінювати тільки у режимі відключення.

Залежності ресурсів. Важливо гарантувати, щоб ресурс налаштування віртуального комп'ютера приводився в робочий стан раніше приведення в робочий стан (запуск) ресурсу віртуального комп'ютера, а переводився в неробочий стан після приведення в неробочий стан (зупинки) останнього.

### 3.2. Grid-системи

Термін «грід-обчислення» з'явився на початку 1990-х років, як метафора, що демонструє можливість простого доступу по потребі до обчислювальних ресурсів (on-demand computing) подібно до електричної мережі (electrical power grid – provides us with computing resources on demand) [Forster, 2001].

Grid визначається як віртуальна організація, яка включає велику кількість гетерогенних ресурсів і дозволяє використовувати всю їх потужність, застосовуючи глобальну інформаційну мережу. Ця система спрямована на гнучкий, безпечний, ресурсно-скоординований доступ до ресурсів (sharing) між множиною членів організації для спільного рішення проблем, використовуючи стратегії ресурсного посередництва.

Поняття ресурсу в grid-системах є абстрактним, але можна таки виділити два основних класи ресурсів: апаратне забезпечення – цикли центрального процесора, дискові сховища, пам'ять, периферійні пристрої, спеціальне обладнання (телескопи, сенсорні мережі), кластери; програмне забезпечення – бази даних, бази знань, додатки, сервіси, результати обчислень.

Область застосування в основному пов’язана з науковими дослідженнями (фізика високих енергій, прогнозування зміни клімату, будування нових

ліків та інше ) - так званим напрямком eScience, сутність якого складається в побудові необхідних сервісів щодо доступу та обробки даних географічно розділеними вченими.

Базовою характеристикою grid-систем є їх здатність віртуалізувати програми, інформацію та інші IT-ресурси такі, як мережі, сервера, пам'ять та настільні комп'ютери. Віртуалізація звільняє програми та інформацію від статичної прив'язки до призначеної фізичної IT-інфраструктури, такій як сервери або пам'ять. Ресурси можуть бути об'єднані в пули, розділені і агреговані незалежно від того, де вони знаходяться,- в одній будівлі або на різних континентах.

Реалізація такої віртуалізації ресурсів стає можливою завдяки впровадженню сервісно-орієнтованої архітектури. Сервіси - це програмні компоненти, максимально абстраговані від специфічних деталей внутрішньої реалізації інших системних служб. Вони описуються за допомогою спеціальних інтерфейсів (контрактів). У кожної служби є ім'я, призначення і політики, пов'язані з безпекою та рівнями служби. Служби можуть бути складені з інших служб або / та користуватися іншими службами для повного завершення деяких специфічних завдань. Призначення служби може бути дуже простим, наприклад, знайти інформацію, або складним - виконати бізнес-процес. Доступ до ресурсу опосередковується сервісом.

Важливу роль в побудові таких систем грають стандарти, які надають можливість інтеграції різноманітних ресурсів в єдиному просторі, та система управління метаданими, яка відповідає за регістрацію/пошук ресурсу за вимогами обчислювальної задачі.

В цілому системою вирішуються наступні основні задачі: пошук необхідних ресурсів, доступ до ресурсу в залежності від його стану, характеристик та прав/пріоритету члена організації і його задачі, координація ресурсів, забезпечення необхідної якості послуги, безпеки та інше.

### 3.2.1. Архітектура Grid-систем

Grid-система має п’яти-шарову архітектуру. Основою такої системи являється ПЗ, що складається із множини сервісів, які за призначенням можна розподілити на 5 основних класів: додатку (application), доступу до ресурсів (collective), з’єднання (connectivity), ресурсів (resources), виконання (fabric). Як правило структура класичної Grid-системи надається у вигляді пісочного годинника, складеного з  шарів (рис.6). Ширина шару визначає множину та різноманіття протоколів, які підтримуються сервісами-додатками на даному шарі. Вузкість серединного шару  підкреслює обмежену кількість протоколів зв’язку в організації системи. Структура, запропонована [Forster, 2001] показана на рис.9,10.

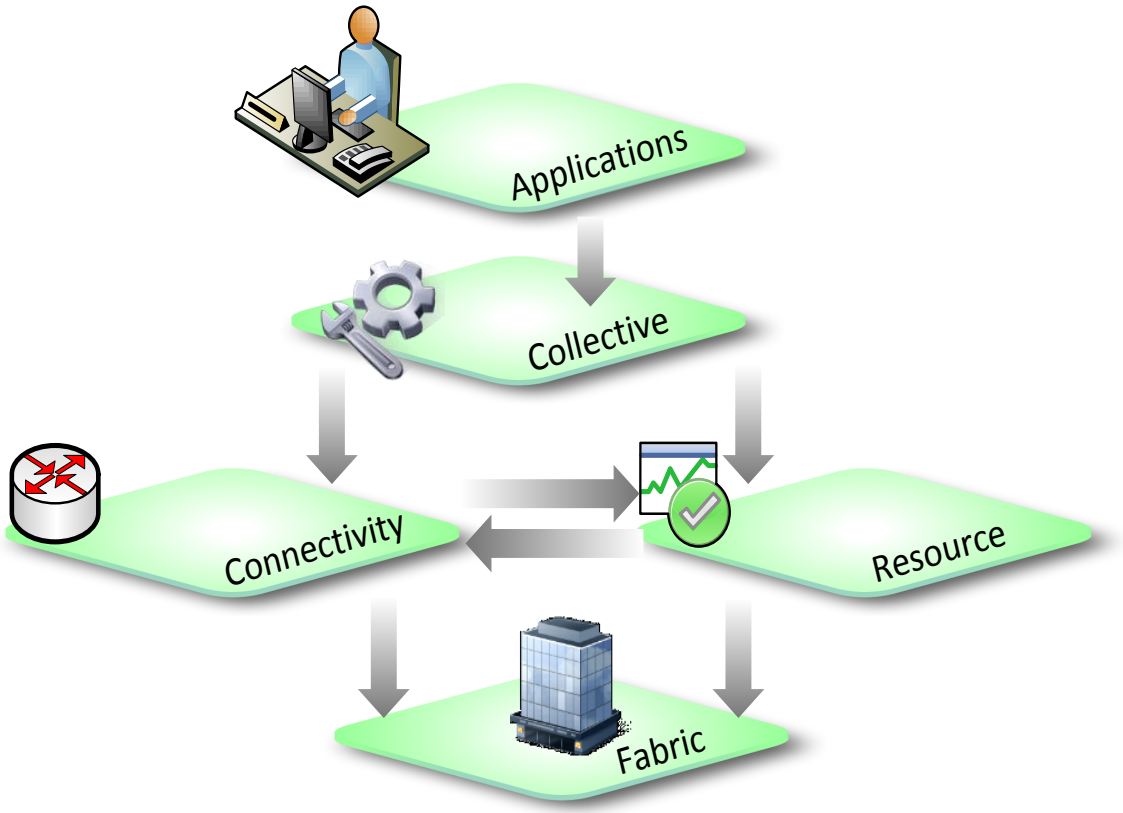



Рис.9. Структура Grid [Forster, 2001].

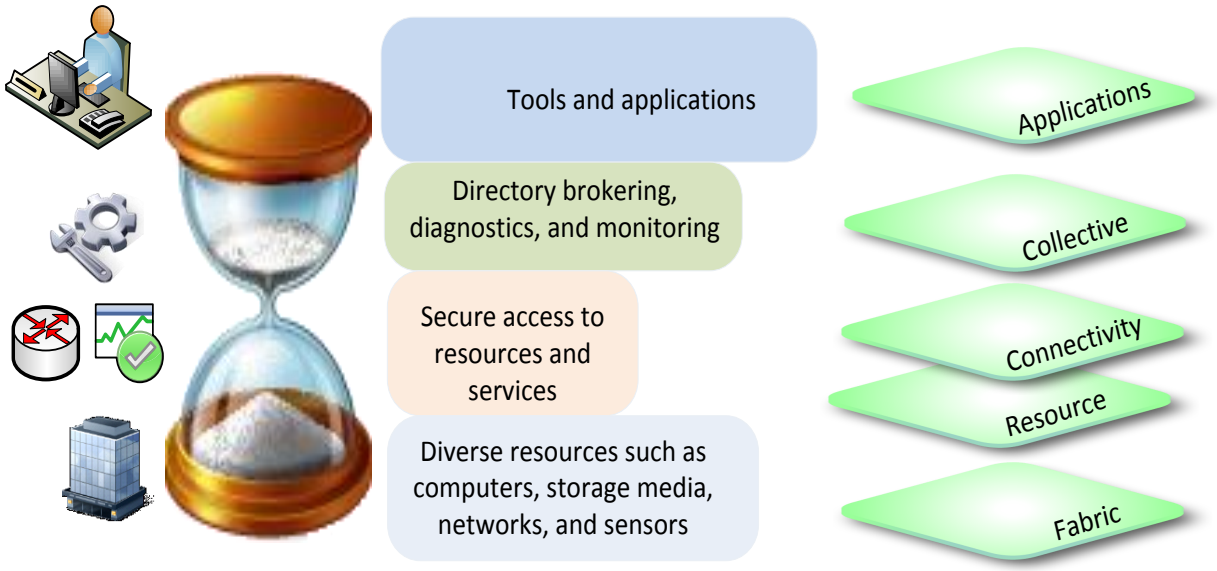



Рис.10. Шари системи Grid [Forster, 2001].

***Рівень взаємодії з ресурсами (Fabric layer)***

Рівень fabric (англ. тканина) відповідає за спільний доступ до ресурсів з застосуванням протоколів Grid. Ресурс може бути логічною сутністю, такою як розподілена файлова система, комп'ютерний кластер чи розподілена комп'ютерна система. В ряді випадків використання ресурсу може включати внутрішні протоколи.

Практично рівень fabric забезпечує інтерфейси для локальних ресурсів на спеціальному сайті системи. *Сайт* в даному випадку означає адміністративну одиницю, яка забезпечує роботу з ресурсом через сервіси. Ці інтерфейси (контракти сервісів) пристосовані до того, щоб дозволяти обмін ресурсами усередині віртуальної організації.

Сервіси цього рівня використовуються для запиту стану та можливостей ресурсу, управління даним ресурсом. Щонайменше ресурси повинні включати: механізми самоаналізу (introspection mechanisms), які дозволяють виявляти їх структуру, стан та можливості (напр., підтримка попереднього резервування, поточне навантаження чи стан черги у випадку ресурсів, які управляються планувальником); механізми управління ресурсами, що забезпечують певний контроль отриманої якості сервісу.

Ресурси зберігання даних включають: механізми розміщення та отримання файлів; механізми для зчитування та запису скорочених варіантів файлу та/або виконання віддалених вибірок даних чи функцій зменшення; механізми контролю даних (дисковий простір, пропускна здатність диску, пропускна здатність мережі, процесор).

Мережні ресурси забезпечують контроль над ресурсами, що є виділеними для мережних передач, використовуються для визначення характеристик та завантаження мережі, контролю помилок при передачах даних.

***Рівень зв'язку (connectivity layer)***

Рівень зв'язку визначає базові проколи комунікації та аутентифікації для здійснення Grid-подібних мережевих транзакцій. Протоколи зв'язку дозволяють забезпечити обмін даними між ресурсами рівня fabric.

Вимоги зв'язку включають у себе *передачу (transport), маршрутизацію (routing) та іменування (naming)*. Мережеві протоколи базуються на відомому стекі протоколів TCP/IP.

Серед найважливіших вимог до сервісів зв'язку визначаються наступні.

*Єдиний вхід в систему (Single sign-on).* Так як користувачі часто хочуть ініціювати обчислення, які використовують велику кількість віддалених ресурсів, користувач повинен мати змогу «увійти» (аутентифікуватися) тільки один раз, а не кожен раз окремо для кожного ресурсу чи при доступі до адміністративного домену.

*Делегування.* Користувач повинен мати змогу надати програмі можливість виконуватися від його імені так, щоб програма могла мати доступ до ресурсів на яких користувач аутентифікувався. Програма також може (вибірково) мати змогу делегувати частину своїх прав іншій програмі: що інколи називають обмеженою делегацією.

*Інтеграція з локальними рішеннями безпеки.* У гетерогенному середовищі кожен сайт чи провайдер ресурсів може вжити значну кількість локальних рішень безпеки. Рішення безпеки Grid повинні мати змогу взаємодіяти із цими локальними рішеннями. Вони не можуть вимагати

масову заміну локальної політики безпеки, але, замість цього, вони повинні дозволяти механізми відображення політик.

*Відносини довіри, орієнтовані на користувача (User-based trust relationships).* Для того, щоб користувач використовував ресурси від багатьох провайдерів разом, система безпеки не повинна вимагати у кожного з провайдерів ресурсів взаємодії один з одним у конфігуруванні сумісного середовища безпеки. Наприклад, якщо користувач, має право використовувати сайти А і В окремо, він також повинен мати змогу використовувати сайти А і В разом, виключаючи при цьому втручання у цей процес адміністраторів безпеки сайтів А і В.

Рішення Grid-безпеки можуть також забезпечувати гнучку підтримку захисту зв'язку (наприклад, контроль над рівнем захисту, захист незалежних одиниць даних для ненадійних протоколів, підтримка транспортних протоколів відмінних від TCP) та дозволяти посередницький контроль над прийняттям рішень щодо аутентифікації, включаючи при цьому можливість забороняти делегацію прав у певних випадках. Наприклад, при необхідності передачі даних між ресурсами або навіть доступу до ресурсу з віддалених точок - протоколи рівню зв'язності будуть містити протоколи безпеки для аутентифікації користувачів та ресурсів.

Слід також відзначити, що у багатьох випадках аутентифікація здійснюється на рівні програм, а не користувачів. У цьому сенсі, делегування прав від користувача до програми – важлива функція, що повинна підтримуватися на рівні зв'язку.

***Рівень ресурсів*** відповідає за створення ресурсу, зупинку процесів, доступ до даних, моніторинг стану, а також за політику додатків (доступ і оплата).

***Рівень колективного використання (Collective layer)*** координує спільне використання різних ресурсів.

Він має справу з управлінням доступом до багатьох ресурсів та складається із сервісів для знаходження (discovery) ресурсів, розподілу та планування

задач для багатьох ресурсів, реплікацій даних та ін. Рівень Collective може складатися з багатьох різних протоколів для широкого спектру послуг, які він може запропонувати віртуальній організації. В якості прикладів розглянемо наступне.

*Служби каталогів* дозволяють членам віртуальної організації запитувати ресурси за ім'ям та атрибутами, такими як тип, доступність чи навантаження.

*Сервіси розподілення, планування та посередництва* дозволяють запитувати виділення одного чи ряду ресурсів для специфічних цілей та планування задач.

*Сервіси реплікації даних* підтримують управління ресурсами з метою оптимізації таких показників, як час відповіді, надійності , ціни тощо.

*Grid-орієнтовані системи програмування* дозволяють використовувати конкретні моделі програмування в Grid-середовищах. Прикладами є Grid-придатні реалізації Message Passing Interface (MPI), одна з яких доступна на сайті http://www.gridmpi.org/.

*Системи робочого потоку* (workflow) забезпечують описання, використання та управління багатоступінчатими, асинхронними, багатокомпонентними робочими потоками.

*Сервіси виявлення* (discovery) програм виявляють та вибирають найкращі реалізації програмного забезпечення.

*Сервіси співробітництва* (collaboration) підтримують координований (синхронний чи асинхронний) обмін інформацією всередині великих співтовариств користувачів.

*Сервіси рівня Collective* повинні вирішувати питання безпеки та звітності.

***Рівень додатків (Application layer)*** складається з додатків, що працюють всередині віртуальної організації та які використовують середовище Grid-обчислень. Додатки побудовані за умовами та за посередництвом сервісів, визначених на будь-якому рівні.

Рівні Collective, Connectivity та Resource вважають серцем grid-системи - ці рівні спільно забезпечують доступ та управління ресурсами, які потенційно розподілені поміж багатьох сайтів. Важливо відзначити що компонент, який вважається “додатком” на одному рівні системи, на практиці може бути складною системою, яка включає підсистеми та компоненти, пов’язані з відповідними апаратними та програмними засобами.

### 3.2.2 Проблеми Grid-систем

Важливою проблемою Grid вважається те, що якщо одна частина програмного забезпечення на вузлі виходить із ладу, інші частини програми, розташовані на інших вузлах також можуть вийти з ладу. Це частково полегшується, якщо вузол забезпечує відмовостійкость, але проблеми все одно існують.

## 3.3. Хмарні обчислення

### 3.3.1. Типи послуг

**Хмара (cloud)** [Gonzalez, 2009] – це тип розподілених систем, що складаються з множини взаємопов’язаних та віртуалізованих комп’ютерів, які динамічно забезпечуються та представляються як один чи більше уніфікованих обчислювальних ресурсів, заснованих на угодах про рівні обслуговування, встановлених через взаємодію між сервіс-провайдером та споживачами. Напрямок **хмарних обчисленнь** (cloud computing) являється представником більш загального напрямку надання обчислювальних послуг за вимогами (on-demand computing.), центральною ідеєю якого являється концентрація кінцевих користувачів-замовників послуг тільки на потребах бізнесу, а не на деталях IT інфраструктури, та специфіці обчислювальних ресурсів (мереж, серверів, систем зберігання даних, програм та послуг). Усі деталі скриті від кінцевих користувачів у хмарі та клієнтам більше не потрібні спеціальні знання для контролю над технологічною інфраструктурою, а потрібно лише своєчасно вносити плату за користування ресурсом (рис.11).

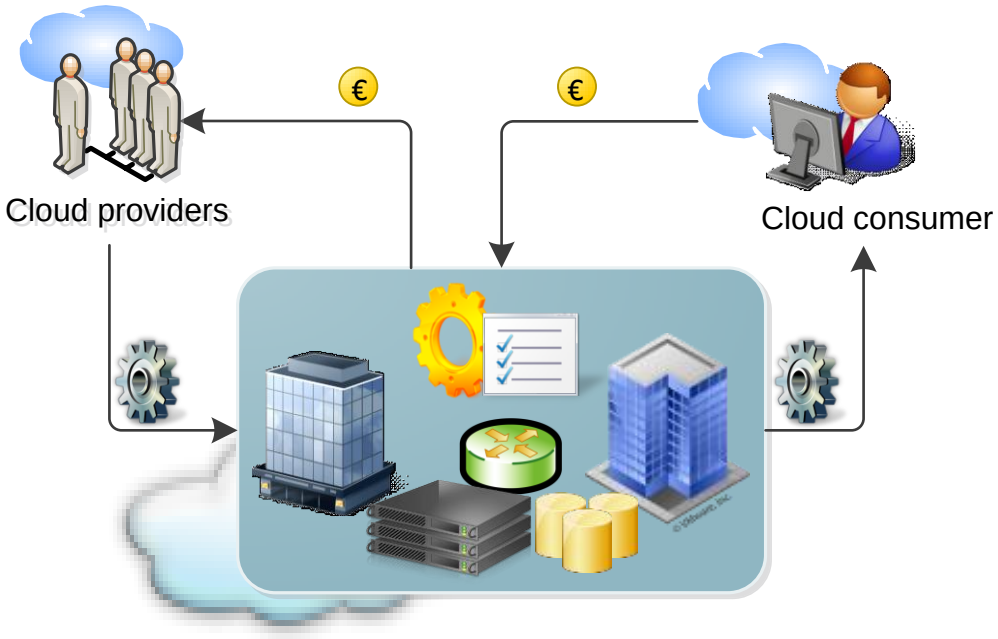



Рис.11. Загальна модель хмарної системи.

Автори більшості існуючих робіт присвячених хмарним обчисленням [Linthicum, 2009],[Redkar, 2011], посилаються на визначення поняття хмарних обчислень, яке було запропоноване Національним інститутом стандартів та технології США ) [NIST, 2011].

Хмарні обчислення - це модель отримання за оплату (pay-per-use) придатного, зручного доступу до загального об'єднання ресурсів з можливістю реконфігурації та налаштування (мереж, серверів, сховищ даних, сервісних додатків), які можуть бути швидко надані або відхилені з мінімальними зусиллями з боку користувачів на взаємодію з провайдером послуги. Оплата береться тільки за користування ресурсом. Модель рекламує доступність (promotes availability). При цьому виділяються п'ять ключових характеристик:

***Самообслуговування за вимогою*** (On-demand self-service). Споживач (consumer) може отримувати за плату послуги (процесорний час, місце на дисках та ін.), виключаючи при цьому взаємодію з провайдером (в однобічному порядку).

***Повсюдний доступ до послуг з використанням мережі*** (Ubiquitous network access). Сервіси є доступними через мережу з використанням стандартних механізмів, які використовуються гетерогенними тонкими та товстими клієнтськими платформами (мобільні пристрої, ноутбуки, кишенькові комп'ютери (Personal Digital Assistant - PDA), інше).

***Незалежність доступу до ресурсів від їх фізичного розташування*** (Location-independent resource pooling). Обчислювальні ресурси мають багаторівневу модель (multitenant model), об'єднуючи фізичні та віртуальні ресурси, що динамічно виділяються замовнику-споживачу, при цьому споживач нічого не знає про реальне розташування ресурсів.

***Швидка зміна об'єму ресурсів у відповідь на зміну вимог від користувача*** (Rapid elasticity), який може отримати потрібні ресурси в любому об'ємі в будь-який час.

***Оплата за використання*** (Pay per use). Використання ресурсів оплачується по однієї з моделей: за використання одиниці ресурсу (metered) – йдеться про вимірювання витраченого ресурсу (дискового простору, мережевого трафіку) ; безкоштовний сервіс (fee-for-service); оплата на основі реклами (advertising-based). В ряді випадків ціна може змінюватися від типу споживача.

Слід відзначити, що значна частина характерних особливостей хмарних обчислень, порівняння з індустрією електрики та використанням публічних, приватних, урядових та суспільних різновидів були досліджені у книзі Дугласа Паркхілла «Проблема обслуговування електронно-обчислювальної техніки», 1966 року [Parkhill, 1966], в якій він запропонував ідею, де обчислювання та накопичення даних могли б використовуватися як публічний сервіс, наданий професіоналами, і кінцеві користувачі не відчули би проблем установки програмного забезпечення та адміністрування.

В якості типового прикладу провайдера «хмарних послуг» слід відзначити Amazon, який отримав декілька десятків тисяч клієнтів: хостінг додатків, резервування та зберігання, доставка контенту, електронна комерція, ви-

сокопродуктивні обчислення, медіа-хостінг, пошукові системи, веб-хостінг, ресурси за потребою. Іншими відомими представниками являються Google Apps (2006), Google App Engine (2009), Sun Cloud (2010), Microsoft Azure (2010). Піковим вважається 2008 рік, коли багато компаній змінило свої IT-інфраструктури на хмарні.

У середині 2008, відома компанія Gartner [Gartner, 2008], яка займається аналізом та досліджуваннями інформаційних технологій, декларувала можливість хмарних обчислень та відзначила, що «організації переходять від власного корпоративного апаратного та програмного забезпечення до сервісно-орієнтованих моделей за необхідністю». Тому «прогнозований перехід до хмарних обчислень…призведе до різкого росту IT-продуктів у деяких областях та значного зменшення у інших».

В [Armbrust, 2010] визначені три основні мотивації хмарних обчислень: вирішення проблеми масштабування системи за вимогами користувачів (the illusion of infinite computing resources on demand); усунення попередніх зобов'язань для користувачів хмари (the elimination of an up-front commitment); можливість платити за використання обчислювальних ресурсів на короткостроковій основі (the ability to pay for use of computing resources on a short-term basis). До цього слід додати якість послуг, надійність, безпеку, доступність, зниження витрат на зміну IT-інфраструктури та її обслуговування, на експерименти та дослідження нових інформаційних технологій. В [Linthicum, 2009] навіть вказується, що хмарні обчислення – це найбезпечніша модель інформатизації підприємств для навколишнього середовища (greenest approach to computing). Аргументами являється консолідація комп'ютерних ресурсів в хмарних центрах з застосуванням сучасних підходів віртуалізації, кластеризації тощо і, як результат, зменшення кількості дата-центрів, в яких сервери значну частину часу простоюють, потребляючи при цьому енергію та виділяючи тепло. Тобто компанія, що використовує хмарні платформи, може вважатися в контексті IT зеленою. Все це дозволяє забезпечити створення віртуальних офісів з еластичною інфраструктурою.

В хмарних обчисленнях визначають три типа послуг (рис.12).

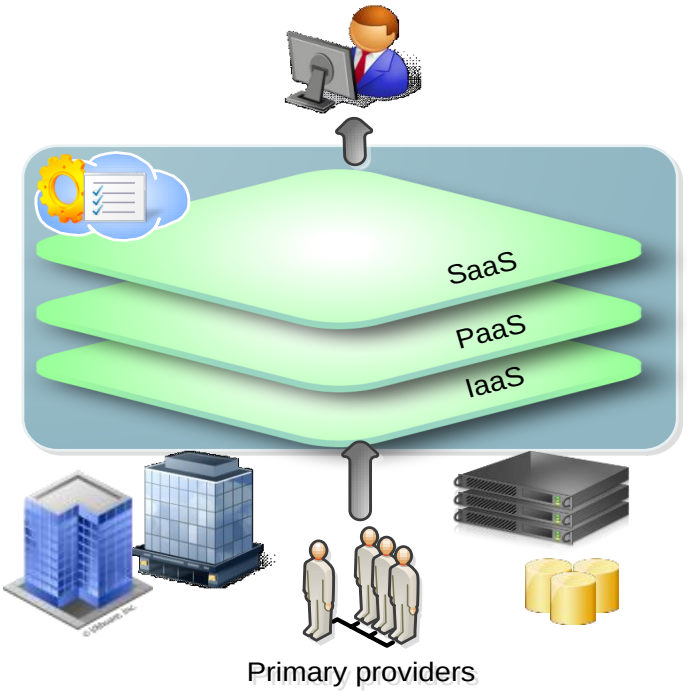



Рис. 12. Три типи послуг хмарних обчислень та їх зв'язок.

***Програмне забезпечення як послуга (Sofware as a Service - SaaS).*** Рівень online сервісів, які надають програму як послугу по Інтернету, усуває необхідність інсталяції та запуску додатку на комп'ютері споживача, спрощують обслуговування і підтримку та забезпечують продуктивність обчислень, а у кінцевому рахунку і бізнесу. Користувач оплачує час використання сервісів (наприклад, Office 365 [MSOnline]), уникаючи питань, пов'язаних з адмініструванням чи ліцензуванням. У сфері бізнесу існують тисячі SaaS доступних на сайті http://www.saas-showplace.com/.

***Платформа як послуга (Platform as a Service - PaaS).*** Сервіси хмарних платформ надають обчислювальну платформу та/або комплект рішень в якості послуги, часто охоплюючи хмарну інфраструктуру та підтримуючі хмарні додатки. Вони полегшують розгортання додатків без витрат та складності покупки та управління базового апаратного та програмного шарів.

***Hardware/Infrastructure as a Service (HaaS/IaaS).*** Сервіси хмарних інфраструктур, які надають комп'ютерну інфраструктуру – зазвичай середовище платформної віртуалізації – в якості послуги, разом із зберіганням даних

та забезпеченням мережі. Замість покупки серверів, програмного забезпечення (далі ПЗ), місця для дата-центрів чи мережевого обладнання, клієнти купують ці ресурсі як повністю замовлений сервіс. Так, наприклад, послуга віртуального серверу включає: сервіси зберігання (класичні API баз даних); мережеві можливості; умови, основані на рівнях обслуговування; енергопостачання, надання повноважень. Оплата при використання здійснюється за часи, дні, місяці користування послугою.

Зв'язок платформ може бути визначеним таким чином: платформа Saas є надбудовою над платформою PaaS, яка в свою чергу включає функціональні можливості платформи IaaS. Використання того чи іншого рівня визначається стратегією компанії.

Клієнт хмари складається з апаратного забезпечення комп'ютера та/або програмного забезпечення, що підключається до «хмари» для роботи додатку. У якості прикладів можна навести певні комп'ютери, телефони та інші пристрої, операційні системи, браузери. Шар серверів складається з продуктів апаратного та програмного забезпечення, які спеціально розроблені для постачання хмарних сервісів, включаючи багатоядерні процесори, хмарно-спеціалізовані операційні системи та змішані пропозиції.

Порівняння функціональних можливостей платформ IaaS, PaaS, SaaS [Redkar, 2011]

Таблиця.2

| | Без хмари | IaaS | PaaS | SaaS |
|---|---|---|---|---|
| Програмні додатки | + | + | + | - |
| Дані | + | + | + | - |
| Ідентифікація користувачів | + | + | + | +- |
| Операційна система та середовища виконання додатків | + | + | - | - |
| Апаратне забезпечення | + | - | - | - |
| Засоби розміщення (hosting facilities) | + | - | - | - |

В реальності для управління ідентифікацією користувачів існують різноманітні сценарії: розділення SaaS, ідентифікації, поєднання (federated service). Також в останні роки виникають додаткові терміни для визначення специфіки сервісів Data as a Service (DaaS), IT as a Service, Security as a Service та інші. Наприклад, в [Linthicum, 2009] наведений наступний перелік додаткових типів сервісів.

*Сховище даних як сервіс (Storage-as-a-service).* Це сервіс, що надає можливість віддаленого використання дискового простору за вимогою користувача для зберігання будь-якої інформації (SkyDrive, DropBox). Безумовно, цей тип послуги є базовим для всіх інших.

*База даних як сервіс (Database-as-a-service - DaaS)* надає можливість оренди системи управління базами даних з віддаленим доступом.

*Інформація як сервіс (Information-as-a-service)* надає можливість користування різноманітною інформацією з використанням спеціального API. Прикладами такого типу послуг може бути: інформація о цінах, перевірка адреси, персони, організації, платоспроможності.

*Процес як сервіс (Process-as-a-service).* Сервіси цього типу спрямовані на зв'язок різноманітних ресурсів для побудови бізнес процесів, які координують взаємодію різноманітних сервісів для досягнення певної мети. Такі процеси можуть бути легко адаптовані до умов бізнесу.

*Інтеграція як сервіс (Integration-as-a-service)* спрямована на інтеграцію гетерогенних додатків, включаючи рівні семантичного посередника, координацію стандартів та інше.

*Безпека як сервіс (Security-as-a-service).* Сервіси спрямовані на забезпечення безпеки доступу, ідентифікації користувачів, передачі інформації.

*Тестування як сервіс (Testing-as-a-service - TaaS).* Сервіси тестування запроваджують користувачеві можливість організації тестування локальних або віддалених систем та сервісів, без витрат на побудову спеціального програмного забезпечення та обладнання.

Клієнт хмари складається з апаратного забезпечення комп'ютера та/або програмного забезпечення, що підключається до «хмари» для роботи додатку. У якості прикладів можна навести певні комп'ютери, телефони та інші пристрої, операційні системи, браузери. Шар серверів складається з продуктів апаратного та програмного забезпечення, які спеціально розроблені для постачання хмарних сервісів, включаючи багатоядерні процесори, хмарно-спеціалізовані операційні системи та змішані пропозиції.

Основними моделями розташування є:

*Приватна хмара (Private cloud).* Інфраструктура хмари доступна виключно для використання однією організацією, що складається з декількох споживачів (наприклад, бізнес - відділів). Хмара може перебувати у власності, управлінні та обслуговуванні однією або декількома організаціями, і може існувати на базі віддаленої платформи провайдера хмари або локально на базі організації.

*Хмара общини (Community cloud).* Хмара призначена виключно для використання визначеною спільнотою споживачів з організацій, які мають спільні проблеми (наприклад, завдання, вимоги безпеки, політики та узгоджень). Хмара може перебувати у власності та управлінні одним чи кількома організаціями в суспільстві і може існувати на базі віддаленої платформи провайдера хмари або на базі організації.

*Загальна хмара (Public cloud).* Хмара для відкритого використання широкої громадськості, яка може перебувати у власності, управлінні бізнес-, академічними, або державними організаціями, або їх комбінацією. Як правило, розташована на базі віддаленої платформи провайдера хмари.

*Гибридна хмара (Hybrid cloud).* Хмара являє собою композицію з двох або більше різних типів хмар (приватних, громадських або державних), які залишаються унікальними, але пов'язані між собою на базі стандартів, що дозволяє забезпечити перенесення даних і додатків (наприклад, для балансування навантаження між хмарами).

### 3.3.2. Хмарна платформа Microsoft Windows Azure

Відомим представником сучасних хмарних платформ являється Microsoft Windows Azure - операційна система з підтримкою хмарних обчислень - і Microsoft Azure Services Platform - платформа для розробки і використання хмарних сервісів на базі Microsoft.NET. Windows Azure забезпечує зберігання, використання і модифікацію даних та запуск програм тільки на комп'ютерах центрів обробки даних Microsoft. Ніякого програмного забезпечення, крім Інтернет-браузера, на комп'ютерах користувачів не потрібно.

З точки зору користувача є дві категорії програм: внутрішні (on-premises applications), які виконуються на комп'ютері користувача; хмарні (cloud applications), які виконуються у середовищі Windows Azure на комп'ютерах центру обробки даних. Доступ до Windows Azure здійснюється через Web-браузер, тому використання сервісів не залежить від ОС клієнтського комп’ютера. Функціонування Windows Azure засноване на Web-сервісах .NET. Для зберігання даних Windows Azure використовує SQL Azure (аналог СУБД MSSQL Server, адаптований до хмарних обчислень). Основними компонентами Windows Azure являються: інтерфейс (fabric), обчислення (compute), пам'ять (storage) і конфігурація (config). Всі компоненти являються web-сервісами. Сервіс «обчислення» виконує користувальні хмарні програми, сервіс «пам'ять» зберігає дані користувача, сервіс «інтерфейс» забезпечує загальні засоби управління додатками, що використовують хмарну платформу.

Проблема масштабування обчислень, яка пов’язана з сервісом «обчислення», вирішується шляхом виконання кожного примірника програми у своїй окремій віртуальній машині. Дані віртуальні машини виконуються в середовищі 64-бітової ОС Windows 2008 Server.

Сервіс «пам'ять» надає користувачеві засоби роботи з даними різної структури - великими бінарними об'єктами (blobs) розміром до 50 Гб, що зберігаються в контейнерах, таблицями (tables) і чергами (queues). Робота зі

структурами даних реалізована на основі ADO.NET - бібліотеками підтримки обробки структурованих даних в .NET.

Сервіс «інтерфейс» реалізований як велика група машин, на кожній з яких працює додаток - агент інтерфейсу (fabric agent). Сервіс «інтерфейс» в цілому управляється ПЗ «контролер інтерфейсу» (fabric controller), який взаємодіє з агентами інтерфейсу, зі сервісом «пам'ять» як із звичайними програмами (тому деталі представлення даних від контролера інтерфейсу приховані). Контролер інтерфейсу управляє кожним хмарним додатком за допомогою конфігураційного файлу.

Як вже говорилося, вся реалізація Windows Azure заснована на надійній та безпечній платформі .NET, виконання програм у якої забезпечується в особливому безпечному режимі (managed execution - кероване виконання). Частина .NET, звана Windows Comminucation Foundation (WCF) та наданий нею механізм сервісів і є основою реалізації платформи Windows Azure. Хмарними сервісами керують дві компоненти - управління доступом (access control) і сервісна шина (service bus), детальний розгляд яких виходить за рамки даного курсу.

Стандартний шлях для розміщення додатку в середовищі Windows Azure детально описаний на сайті http://www.microsoft.com/BizSpark/Azure/ HowToDeployAzureApp.aspx. По-перше, необхідно мати обліковий запис Microsoft. Далі необхідно створити обліковий запис (account) для платформи Windows Azure (https://windows.azure.com/ NoAccount.aspx). Microsoft надає можливість використання Windows Azure в продовж 90 днів. Для підтвердження особистості користувач надає Windows Live ID та номер кредитної карти. Після закінчення пробного періоду клієнт не зобов'язаний оплачувати подальше використання.

### 3.3.3. Проблеми впровадження хмарних обчисленнь

*Конфіденційність.* Хмарна модель критикувалася захисниками конфіденційності за значну простоту, при якій компанії, що утримують хмарні

сервіси, контролюють та можуть перевіряти за своїм бажанням зв'язок та дані, що зберігаються між користувачем та головною компанією. Прикладом є секретна програма NSA, працююча з AT&T та Verizon, яка записала більше 10 мільйонів телефонних дзвінків між американськими громадянами.

*Відкриті стандарти.* Більшість провайдерів виставляють API, які зазвичай добре документовані (часто за ліцензією Creative Commons), але унікальні за своїм виконанням, а від так не інтероперабільні. Деякі поставники перейняли API інших, також у розробці є декілька відкритих стандартів із метою надання інтероперабельності та портабельності.

*Безпека.* Відносна безпека сервісів хмарних обчислень — це суперечливе питання, що може затримувати їх впровадження. Проблеми, що перешкоджають прийняттю хмарних обчислень у великій мірі виникають через неспокій приватних та публічних секторів, які оточує зовнішнє управління безпеко-орієнтованими сервісами. Це є великим стимулом провайдерів послуг хмарних обчислень надавати пріоритет створенню та утриманню сильного управління безпечними сервісами. Проблеми безпеки були класифіковані на такі: доступ до вразливих даних, відокремлення даних, конфіденційність, обробки помилок, відновлення, звітність, внутрішні зловмисники, безпеки консолі управління, контроль облікових записів. Вирішення різноманітних проблем полягають у криптографії, а саме інфраструктурі відкритих ключів (public key infrastructure - PKI), використанні декількох хмарних провайдерів, стандартизації API, покращенні підтримки віртуальних машин та юридичній підтримці.

### 3.3.4. Порівняння Grid та хмарних обчислень

Обидва типи систем являються представниками напрямку «обчислення на потребу» (on-demand computing). Але системи Grid більш орієнтовані на створення віртуальних організацій з гетерогенними ресурсами, сервісами обробки та доступу співтовариств географічно розподілених членів. На відміну від цього хмарні обчислення орієнтовні на продаж якісних послуг кінцевим

користувачам, при цьому ресурси уніфіковані та належать постачальнику послуг.

Хмарні обчислення та Grid-обчислення орієнтовані на забезпечення масштабованності системи, яка досягається через балансування навантаження складових додатку, працюючих роздільно на множині операційних систем та зв'язаних через веб-сервіси. Процесорний час, пропускна здатність мережі та інші ресурси виділяються за вимогами та статусом користувача. Обидва типи обчислень полагаються на паралельну, розподілену обробку, багатозадачність. Розподілення ресурсів між великим пулом користувачів допомагає зменшити витрати на інфраструктуру та пікову завантаженість. Хмарні та Grid обчислення забезпечують угоди про рівні обслуговування для гарантованого доступного часу безвідмовної роботи. Але підходи до забезпечення якості послуги також відрізняються – хмара покладається на віртуалізацію машин, серверів з відповідною технікою кластерізації, виділенням трьох базових рівнів послуги, яскраво вираженою комерційною складовою; грід базується на ресурсах, запроваджених вільними членами організації, які не пов’язані зобов’язаннями, тому здається неможливим планувати, контролювати якість і навіть здійснення послуги.

## 4. Розподілені інформаційні системи

***Система*** – це множина взаємопов'язаних елементів (interdependent elements), яка відокремлена від середовища і взаємодіє з ним як ціле [Peregudov, 1989].

За визначенням [Haas, 2007] ***інформаційна система*** – це система, яка з певною метою управляє обробкою множини фактів, що відноситься до деякої області знань (a system that manages facts about some domain for a specific purpose). При цьому факти означають актуальні дані, що обробляються системою, а область знань – це область міркування (universe of discourse), яка задає обмеження на множину об’єктів до яких ці факти мають відношення [Bool, 1854]. Управління даними сфокусовано на питаннях ефективності та

надійності обробки великих масивів даних, але управління інформацією має на увазі управління складною інтерпретацією даних з використанням семантичних аспектів обробки. Безумовно ці поняття пов'язані – інформаційна обробка базується на обробці даних. Інформаційні комп'ютерні системи призначені для підвищення ефективності та результативності роботи організації. Ступень такого підвищення залежить від функціональних можливостей (capabilities) інформаційної системи та характеристик організації (системи праці, співробітників, методологій розробки та впровадження) [Silver, 1995].

Основними типами інформаційних систем являються: системи обробки транзакцій (transaction processing systems), системи підтримки прийняття рішень (decision support systems), системи управління знаннями (knowledge management systems), системи управління базами даних (database management systems), офісні інформаційні системи (office information systems).

***Розподілені інформаційні системи*** – клас інформаційних систем, в якому компоненти системи (дані та додатки) фізично розподілені по різним вузлам (комп'ютерам). Мотивами такого розподілення є: паралельна обробка, покращення масштабованості, взаємодія вже існуючих систем (за даними на 2005 рік 70% розробок пов'язано з побудовою нових систем на базі інтеграції вже існуючих) та інше. За визначенням [Tanenbaum, 2006] клас розподілених інформаційних систем включає різновиди програмних додатків із широким застосуванням комп'ютерних мереж (networked application).

В даній роботі розглянемо два важливих типи розподілених інформаційних систем: системи обробки розподілених транзакцій (distributed transaction processing systems) та системи інтеграції корпоративних додатків (enterprise application integration).

### 4.1. Системи обробки транзакцій

З точки зору бізнесу **транзакція** (business transaction – досл. ділова операція) – це взаємодія між підприємствами або підприємством і клієнтом, яка включає процедуру обміну сутностями (where something is exchanged) [Bernstein, 2009]. У якості таких сутностей можуть виступати – гроші, продукти, інформація, сервісні запити. Важливою властивістю транзакції являється її *складність*, тобто означена процедура обміну може складатися з послідовності певного числа елементарних шагів-операцій. Успішне завершення транзакції обумовлено успішним завершенням усіх її складових операцій. Цей принцип отримав назву «все або нічого» (all-or-nothing) і являється базовим в системах обробки транзакцій. Для підвищення швидкості та надійності, зменшення ціни обробки бізнес транзакцій застосовуються комп'ютерні системи, зв'язок між взаємодіючими сторонами здійснюється на базі комп'ютерної мережі, як правило, на базі Інтернету.

Комп'ютерні системи, які спрямовані на обробку транзакцій називаються системами обробки транзакцій (transaction processing systems – TP-systems). В [Bernstein, 2009] підкреслюється, що з точки зору комп'ютерної системи он-лайн транзакція (on-line transaction) – це виконання заданої програми, а не повідомлення, або сама програма. В [Bolton, 2010] поняття транзакції бази даних було визначено як одиниця роботи (unit of work), яка складається з декількох команд читання з або запису до бази даних. Он-лайн транзакція, як правило, приймає участь у виконанні бізнес функції, але не виконує її. Як правило обробка транзакції пов'язана з діями користувача (on-line user), але деякі системи не пов'язані з користувачами – наприклад, система запису повідомлень зі супутнику. Деякі програми обробки транзакцій працюють у режимі off-line тому, що час, за який виконується множина операцій, перевищує час допустимий для очікування користувачем. Першим додатком для обробки он-лайн транзакцій вважається SABRE (Semi-Automatic Business-Related Environment) - система

резервування польотів, що була розроблена в 1960-их роках спільними зусиллями компаній IBM та American Airlines.

У багатьох випадках мережний додаток складається із серверу, що запускає цей додаток та робить цей додаток доступним для віддалених програм, що називаються клієнтами. Часто серверна частина пов'язана з використанням бази даних, яка працює під управлінням спеціальних сервісів, що складають систему управління базами даних. Такі клієнти можуть відправляти запити на сервер для виконання деякої операції, після чого, назад буде відіслано відповідь. Інтеграція на системному рівні дозволяє клієнтам об'єднати декілька запитів, можливо для різних серверів, у єдиний більший запит та виконати їх як розподілену транзакцію, сутність якої полягає у виконанні складових підзапитів, які складають тіло транзакції. У якості таких запитів можуть виступати і системні запити, і запити бібліотечних процедур, і запити до бази даних. Для спрощення сприйняття матеріалу подальший розгляд буде сконцентрований переважно на особливостях обробки транзакцій на прикладі додатків, які орієнтовані на роботу з базами даних (database applications).

Створення програм обробки транзакцій вимагає введення спеціальних примітивів, що повинні бути забезпечені базовою розподіленою системою (underlying distributed system) або системою середовища виконання (language runtime system). Точний перелік цих примітивів залежить від того, які типи об'єктів використовуються у транзакції. Так у поштовій системі можуть бути введені примітиви для відправлення, отримання, та пересилки пошти. В банківській сфері – типовими примітивами можуть бути команди читання залишку, переведення грошей з одного рахунку на інший та інш. В системах управління базами даних – конструкції мови SQL. Виклики віддалених процедур (remote procedure call – RPC) також можуть бути інкапсульовані у транзакцію.

### 4.1.1. Архітектура системи обробки транзакцій

**Система обробки транзакцій** – комп'ютерна система (тандем апаратного та програмного забезпечення), спрямована на обробку транзакцій. Вона складається із наступних основних компонентів.

1. Пристрій кінцевого користувача (end-user device). Кінцевий користувач ініціює виконання бізнес-транзакції (наприклад, користувач банку). В якості пристроїв, які забезпечують інтерфейс користувача з ядром системи можуть виступати спеціалізовані пристрої (dumb device – німі пристрої, які здатні відображати та пересилати дані до віддаленої системи-сервера обробки транзакцій), наприклад банкомат чи касовий апарат. Але в більшості випадків - це персональний комп'ютер підключений до мережі Інтернет, який працює під керуванням багатозадачної операційної системи та забезпечує доступ до системи-сервера за допомогою інтерфейсної програми (front-end program) .

2. Інтерфейсна програма (front-end program). Інтерфейсна програма – це додаток, встановлений на пристрої кінцевого користувача, який надає змогу користувачеві вибрати та запустити на виконання процедуру певної транзакції. Часто такий інтерфейс забезпечується програмою, що розташована на веб-сервері і яка взаємодіє з браузером по протоколу HTTP. Дана програма перевіряє дані введені користувачем (validates the user's input) та відправляє запит до модуля, який відповідає за обробку транзакції.

3. Контролер запитів (request controller). Модуль контролеру (монітору) запитів відповідає за отримання запитів від інтерфейсних програм та відображення кожного запиту користувача на один або декілька запитів до програм-обробників транзакцій. У разі призначення декількох програм-обробників контролер проводить моніторинг стану запиту (tracks the state of the request).

4. Сервер обробки транзакцій (transaction server). Сервер обробки – це процес, який виконує обробку транзакції, відповідно до заданих бізнес-

правил. Як правило така обробка пов’язана з виконанням операцій читання, додавання та оновлення інформації в системі управління базами даних, викликом інших програм та поверненням результатів обробки модулю, який надіслав запит.

5. Система управління базами даних (database system) – множина програмних компонентів-сервісів, які управляють сховищами загальних даних (shared data). Такі дані часто називаються стійкими (persistent, durable). Вони зберігаються на жорстких дисках та складають основу додатків, що відповідають за інформаційну підтримку бізнес-функцій.

### 4.1.2. Властивості транзакцій

Важливою властивістю транзакції являється вимога виконання або всіх, або жодної із складових команд, які утворюють тіло транзакції. Ця властивість “все-або-нічого” – одна із чотирьох характерних вимог до класичних систем обробки транзакцій. Ці вимоги (властивості транзакцій) часто називають за їх початковими літерами ACID (Atomic, Consistent, Isolated, Durable), виконання яких гарантує консистентність та можливість відновлення баз даних майже у разі виходу з ладу апаратного забезпечення. Порушення цих правил приводить до проблем, які будуть розглядатися далі. Важливо також відзначити гарантування правильного виконання не тільки явних складних транзакцій (explicit multistatement transactions), але й прихованих транзакцій (implicit transactions), що мають відношення до виконання звичайних з точки зору користувача SQL-команд, які в реальності зводяться до набору простіших операцій, пов’язаних з обробкою деякої області бази даних (набору рядків). Прикладом команди з прихованою транзакцією являється команда UPDATE, яка модифікує 10 рядків – якщо зміна одного з рядків викличе помилку, то команда повинна вважатися невиконаною, а результати її виконання повинні бути анульовані. Розглянемо властивості транзакцій.

1. ***Атомарність*** (*Atomic*). Для зовнішнього середовища, транзакції здаються нероздільними. Поки транзакція знаходиться у процесі виконання, інші процеси не можуть бачити жоден із проміжних станів.

2. ***Консистентність*** *(Consistent).* Транзакція повинна не порушувати обмеження (constraints) або правила (rules), які визначені в системі для даних. Інакше кажучи, транзакція не порушує системні інваріанти, цілісність використовуємих даних (referential integrity).

3. ***Ізольованість*** (*Isolated or serializable*). Паралельні транзакції не заважають одна одній. Якщо дві чи більше транзакцій працюють одночасно, для кожної з них та і для інших процесів, кінцевий результат виглядає так, наче усі транзакції працювали послідовно у певному порядку.

4. ***Надійність*** (*Durable*). Коли транзакція підтверджена, зміни лишаються постійними. Це означає, що коли транзакція підтверджується результати стають постійними. Жодна відмова в системі після підтвердження не може повернути результати чи спричинити їх втрату.

Звичайна транзакція пов’язана з одним ресурсом (базою даних, файлом, чергою повідомлень). Складена транзакція (nested transaction) побудована із декількох підтранзакцій (subtransaction), які можуть бути простими або складеними (рис.13). Транзакція вищого рівня може включати дочірні, які працюють паралельно з різними ресурсами та на різних машинах. Кожна з підтранзакцій може також виконувати одну чи більше підтранзакцій. Основними мотивами введення та системної підтримки конструкції розподіленої транзакції являється покращення ефективності роботи системи та спрощення програмування. Коли певна транзакція чи підтранзакція починає виконуватися, їй надається приватна копія усіх даних. Якщо вона відміняється (abort, rollback), її приватна область видаляється. Якщо вона підтверджується (commit) – її приватна область замінює область актуальних даних.

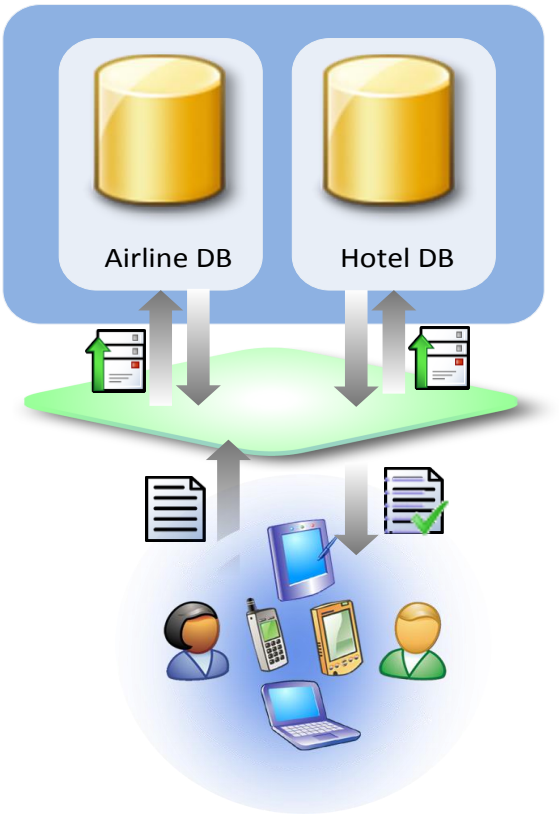



Рис. 13. Вкладені транзакції

Вкладені транзакції важливі у розподілених системах для забезпечення природного шляху розподілення транзакції поміж багатьох машин. Вони слідують логічному розподіленню роботи початкової транзакції. Наприклад, транзакція для планування поїздки, за якою необхідно зарезервувати три різні польоти, може бути логічно розділена на три підтранзакції. Кожна із підтранзакцій може керуватися роздільно та незалежно одна від одної.

На початку розвитку корпоративних систем, компонент, що керував розподіленою (або вкладеною) транзакцією, формував ядро для інтеграції додатків на рівні серверу чи бази даних. Цей компонент називається монітором, менеджером або координатором обробки транзакцій. Його головними задачами є – дозволити додатку отримати доступ до декількох серверів/баз даних, пропонуючи йому транзакційну модель програмування; визначити порядок запуску підтранзакцій на базі графу їх залежності; забезпечити контроль за виконанням частин розподіленої транзакції. Забезпечення коректної роботи транзакції здійснюється на базі двохфазного протоколу.

### 4.1.3. Протокол двохфазного підтвердження

У разі обробки розподіленої транзакції, яка пов'язана з оновленням даних декількох баз даних, знову виникає необхідність контролю атомарності. Проблема полягає в тому, що підтранзакції розподіленої транзакції обробляються незалежними додатками і частина підтранзакцій може бути виконана успішно, а частина відмінена. У якості приклада можна навести транзакцію, яка запускає паралельно декілька підтранзакцій, і одна із них робить свої результати видимими для батьківської транзакції. Після подальших обчислень батьківська транзакція перериває роботу, відновлюючи систему до того стану, який вона мала перед тим як транзакція розпочалася. Отже, результати підтранзакції, яка була успішно виконана повинні бути анульовані.

Іншою проблемою являється переривання обробки підтранзакції у разі відмови системи. У цьому випадку ми не маємо однозначної відповіді про результати обробки підтранзакції. Тому виникає питання доцільності очікування відновлення роботи системи зі зберіганням при цьому результатів інших підтранзакцій. Це вимагає від обробників транзакцій (менеджерів ресурсів) зберігання копій результатів, проміжних результатів на диску; а від менеджера розподілених транзакцій реалізації політик (наборів правил) для вирішення проблемних ситуацій.

Вирішення проблеми атомарності базується на застосуванні так званого протоколу подвійного підтвердження (two-phase commit – 2PC), за виконання якого відповідає модуль менеджера транзакцій (transaction manager). Кожен з менеджерів ресурсів після отримання від менеджера (координатора) розподіленої транзакції команди на обробку транзакції, здійснює копіювання необхідних даних та починає їх обробку, після завершення якої зберігає результати, повідомляє менеджера про успішне або неуспішне виконання. На цьому завершується перша фаза виконання розподіленої транзакції – дані підготовлені до відновлення (prepared to commit) (рис.14). Далі менеджер ресурсу очікує підтвердження від менеджеру розподіленої транзакції. У разі підт-

вердження розподіленої транзакції всі менеджери локально підтверджують результати (рис. 15).

З обробкою транзакцій пов'язані питання швидкості, ефективності обробки, але в даній роботі ми сфокусуємо увагу на особливостях рівнів ізольованості транзакцій БД та засобах фірми Microsoft для підтримки виконання розподілених транзакцій з менеджерами ресурсів різних типів.

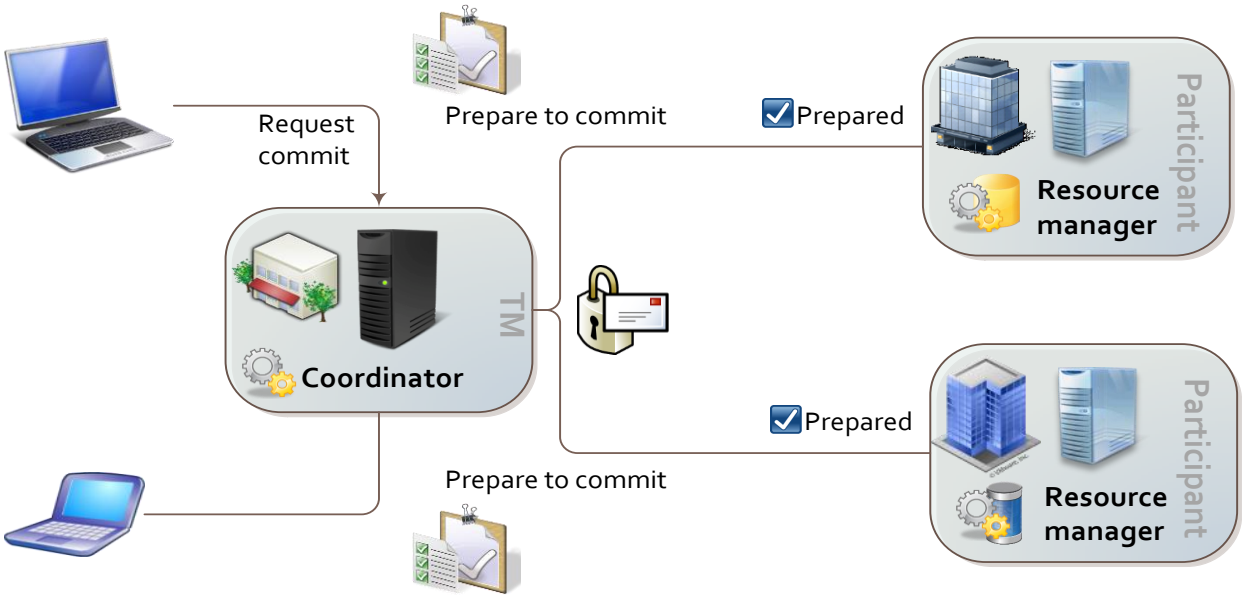



Рис. 14. Перша фаза двохфазового протоколу підтвердження розподіленої транзакції

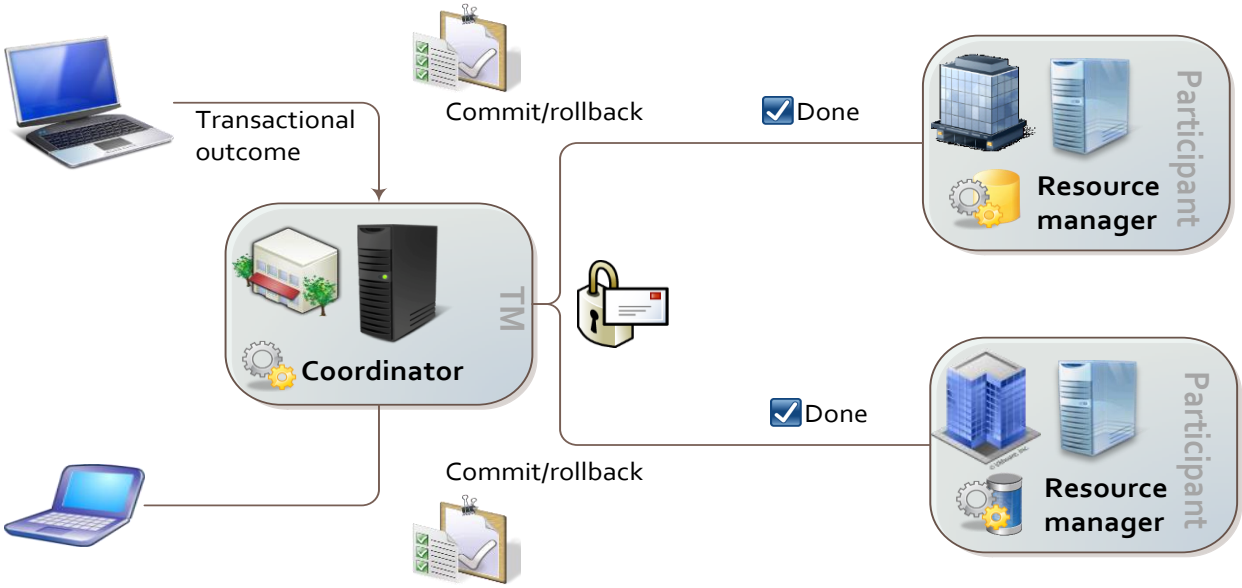



Рис.15. Друга фаза двохфазового протоколу

### 4.1.4. Сутність блокування

Механізм блокування базується на маркуванні області даних менеджером транзакцій, роблячи їх недоступними для інших транзакцій, доки перша транзакція не підтвердиться чи відміниться. Блок завжди повинен бути встановлений перед обробкою даних, включаючи дані, що зчитуються, але не змінюються. Складні транзакції зазвичай вимагають великої кількості блокувань, що призводить до серйозних накладних витрат, а також до блокування інших транзакцій. Наприклад, якщо користувач А працює з транзакцією, що повинна прочитати рядок даних, які користувач В хоче змінити, користувач В чекає, завершення транзакція користувача А.

Альтернативою блокування є управління паралельним доступом (concurrency) за допомогою *багатоверсійності* (MultiVersion Concurrency Control – MVCC), у якому база даних забезпечує кожній зчитувальній транзакції попередню, незмінену версію даних, що модифікуються в даний момент іншою транзакцією. Це дозволяє зчитувачам даних оперувати даними без необхідності блоків, тобто транзакції запису не блокують транзакції зчитування, і навпаки (readers never block writers, and writers never block readers). Користувач А отримує актуальне представлення даних, навіть якщо інші користувачі змінюють ці дані. Ряд продуктів підтримують такий тип блокування: SQL Server (з версії 2005 має установку на рівень ізольованості SET READ_COMMITTED_SNAPSHOT ON), Oracle (версії 8 і більше), MySQL 5 (тільки InnoDB tables), PostgreSQL, Firebird, Informix.

### 4.1.5. Рівні ізольованості

Більшість СУБД пропонують певну кількість рівнів ізольованості транзакцій, що контролюють ступінь блокування даних, які обробляються в транзакції. Існування багатьох рівнів ізольованості обумовлене спробою зробити систему більш продуктивною за рахунок зменшення вимог до блокування записів і часткового порушення базового принципу ізольованості обробки транзакцій (ACID). Користуючись полегшеними рівнями ізольованості слід

ясно усвідомлювати ризик і, так звані, «проблеми залежності» (dependency problems, anomalies або consistency problems), які можуть виникати внаслідок такого полегшення. До речі, не всі автори вважають такі порушення проблемами. Так в [Delaney, 2009] вони вважаються допустими (за вибором розробника) поведінками (behaviors). Навпаки, користуючись жорстким методом блокування слід також усвідомлювати доцільність його застосування та негативні наслідки, які виражаються в збільшенні простоїв системи, зниженні продуктивності. Рівні ізольованості, що визначені стандартами ANSI/ISO SQL, підтримуються усіма відомими провайдерами баз даних. Розглянемо них.

*SERIALIZABLE.* Найвищий рівень ізольованості. Паралельне виконання або суміщення деяких фаз транзакцій виглядає так, наче усі транзакції працювали послідовно у визначеному порядку. При реалізаціях СУБД із контролем паралелізму на основі блокування, послідовність вимагає звільняти блоки зчитування та запису у кінці транзакції. Також повинні бути запитані можливості блокування області, коли запит SELECT використовує розширений вираз WHERE.

*REPEATABLE READS.* На цьому рівні ізольованості працюють реалізації СУБД із управлінням конкурентним доступом, тримають блокування запису та зчитування до кінця транзакції. Однак, немає управління блокуваннями області, так що може виникати явище phantom reads (див. нижче).

READ COMMITTED. На цьому рівні ізольованості працюють реалізації СУБД із управлінням конкурентним доступом, тримають блокування запису та зчитування до кінця транзакції, але блокування зчитування знімаються після того, як виконався запит SELECT (тому, на цьому рівні може виникнути феномен on-repeatable reads).

*READ UNCOMMITTED.* Це найнижчий рівень ізольованості. На цьому рівні дозволені *dirty reads*, тому одна транзакція може бачити ще не підтверджені зміни, зроблені іншою транзакцією.

### 4.1.6. Проблеми залежності транзакцій

ANSI/ISO стандарт SQL 92 посилається на три проблеми (феномени зчитування), коли перша транзакція зчитує дані, які друга транзакція могла змінити. У наступних прикладах мають місце дві транзакції. В першій виконується Запит 1. Потім, в другій транзакції виконується та підтверджується Запит 2. По завершенню, в першій транзакції знову виконується Запит 1.

*Актуалізація непідтверджених даних (dirty reads).* Явище dirty reads виникає, коли транзакції дозволяється зчитувати дані із області (рядку), що була модифікована іншою працюючою транзакцією, та не була підтверджена. Dirty reads працюють схоже із non-repeatable reads; однак, другій транзакції не потрібно буде підтверджуватися, щоб перший запит повернув інший результат. Важлива річ, яку потрібно запобігати на рівні ізольованості READ UNCOMMITTED — це оновлення, які з'являються не по порядку в результатах; тобто, попередні оновлення можуть з'являтися у результуючому наборі перед пізнішими.

Наприклад, транзакція 2 змінює рядок, але не підтверджує зміни. Потім транзакція 1 зчитує непідтверджені дані. Тепер, якщо транзакція 2 відмінить зміни (вже зчитані та актуалізовані транзакцією 1), результат транзакції 1 не буде вірним. Наприклад,

| ***Transaction 1*** | ***Transaction 2*** |
|---|---|
| */* Запит 1 */*<br>**SELECT** * **FROM** users **WHERE** id = 1; | |
| | */* Запит 2 */*<br>**UPDATE** users **SET** age = 21 **WHERE** id = 1;<br>*/* Підтвердження немає */* |
| */* Запит 1 */*<br>**SELECT** * **FROM** users **WHERE** id = 1; | |
| | **ROLLBACK**; */* lock-based DIRTY READ */* |

*Дві однакові вибірки в рамках однієї транзакції повертають різні результати (Non-repeatable reads).* Дана проблема може виникати при методі управління конкурентним доступом до області даних. Дані, які зчитуються операцією SELECT не блокуються, або блокування області знімається коли операція SELECT виконана. При використанні в СУБД методу контролю конкурентним доступом на основі багатоверсійності (MVCC) дана проблема може виникати, коли послаблена вимога щодо відміни результатів транзакції, на яку вплинув конфлікт підтвердження. Наприклад

| ***Transaction 1*** | ***Transaction 2*** |
|---|---|
| */* Query 1 */*<br>**SELECT** * **FROM** users **WHERE** id = 1; | |
| | */* Query 2 */*<br>**UPDATE** users **SET** age = 21 **WHERE** id = 1;<br>COMMIT; */* *in multiversion concurrency control, or lock-based READ COMMITTED* */* |
| */* Query 1 */*<br>**SELECT** * **FROM** users **WHERE** id = 1;<br>COMMIT; */* *lock-based REPEATABLE READ* */* | |

У цьому прикладі транзакція 2 успішно підтверджується, що викликає оновлення бази даних, і транзакція 1, яка у черзі стояла попереду і повинна закінчити виконання перед другою транзакцією у разі існування залежностей отримує дані, які були змінені більш швидкою транзакцією. На рівнях ізольованості SERIALIZABLE та REPEATABLE READ СУБД повертає старе значення. На рівнях READ COMMITTED та READ UNCOMMITTED СУБД повертає змінене значення.

Існують дві основні стратегії, що використовуються для запобігання явищу non-repeatable reads. Перша – у разі виявлення залежності відкласти

виконання транзакції 2 до закінчення виконання транзакції 1. Цей метод використовується, коли використовується блокування, та відпрацьовує залежні транзакції послідовно. У іншій стратегії, яка базується на контролі за конкурентним доступом на основі багатоверсійності, транзакції 2 дозволяється підтверджуватися першою, що забезпечує кращий паралелізм. Однак, транзакція 1, яка розпочалася до транзакції 2, повинна продовжувати працювати з минулою версією даних — знімком на той момент, коли вона розпочалася. Коли транзакція 1 зрештою намагається підтвердитися, СУБД перевіряє, чи може буде результат фіксування транзакції 1 еквівалентним до розкладу T1, T2. Якщо так, тоді транзакція 1 може продовжуватися, якщо ні – транзакція 1 повинна бути відмінена з помилкою порушення порядку.

Використовуючи метод контролю за конкурентним доступом на основі блокування у режимі ізольованості REPEATABLE READ, рядок із id=1 заблокується, від так блокуючи Запит 2, поки перша транзакція не завершить роботу. При управлінні конкурентним доступом за допомогою багатоверсійності, на рівні ізольованості SERIALIZABLE, обидва запити SELECT бачать знімок бази даних, зроблений на початку транзакції 1. Тому, якщо транзакція 1( триває більше часу) буде намагатися також оновити дані, які були змінені транзакцією 2 – виникне помилка порядку виконання, результати транзакції 1 будуть анульовані і транзакція 1 буде поставлена у чергу до виконання. На рівні ізольованості READ COMMITTED, кожен запит бачить знімок області бази даних, зроблений на початку кожного запиту – помилка порядку виконання не виникає і транзакції 1 не потрібно буде повторюватися.

***Фантоми або фантомні зчитування (Phantom reads).*** Явище phantom reads виникає, коли під час транзакції виконуються два ідентичних запити і колекція рядків, що повертаються другим запитом відрізняється від результату першого запиту.

Це може виникати, коли при виконанні операції SELECT ... WHERE не здійснюється блокування області. Аномалія phantom reads — це особливий випадок Non-repeatable reads, коли транзакція 1 повторює запит SELECT ...

WHERE, і посеред обох операцій транзакція 2 створює (напр. INSERT) новий рядок (у цільовій таблиці), який входить в область даних заданих умовою WHERE.

| ***Transaction 1*** | ***Transaction 2*** |
|---|---|
| */* Запит 1 */*<br>**SELECT** * **FROM** users<br>**WHERE** age **BETWEEN** 10 **AND** 30; | |
| | */* Запит 2 */*<br>**INSERT INTO** users **VALUES** ( 3, 'Bob', 27 );<br>COMMIT; |
| */* Запит 1 */*<br>**SELECT** * **FROM** users<br>**WHERE** age **BETWEEN** 10 **AND** 30; | |

На рівні ізольованості SERIALIZABLE у випадку спроби виконання запиту 2 паралельно з виконанням транзакції 1 всі записи із полем age у діапазоні від 10 до 30 заблокуються, що викличе блокування запиту 2 до того моменту, поки транзакція 1 не завершиться. У режимі REPEATABLE READ область не заблокується, що дозволить додати новий запис у область, яка використовується транзакцією 1, а другому виконанню запиту 1 це дозволить включити новий рядок у свій результат. В таблиці 3 показаний зв'язок проблем (феноменів) залежності та рівнів ізольованості транзакцій. Знак “+” означає присутність феномену на відповідному рівні.

Зв'язок рівнів ізольованості з проблемами залежності

Таблиця 3

| Рівень ізольованості | Проблеми | | |
|---|---|---|---|
| | Dirty reads | Non-repeatable reads | Phantoms |
| Read Uncommitted | + | + | + |
| Read Committed | - | + | + |
| Repeatable Read | - | - | + |
| Serializable | - | - | - |

## 4.2. Особливості обробки транзакцій в Microsoft SQLServer

Розглянемо практичні аспекти обробки транзакцій на прикладі системи Microsoft SQLServer 2008.

Ядро SQL Server можна представити як композицію двох базових підсистем: *обробника запитів* ( Relational Engine, query processor), який відповідає за оптимізацію та виконання запитів; *менеджера сховища* (Storage Engine), що відповідає за управління вводом/виводом фактів в базу даних.

***Обробник запитів*** (Relational Engine) включає: модуль розбору SQL-команд (Command Parser), який відповідає за перевірку синтаксису команд-запитів та підготовку дерева запитів (query trees); оптимізатор запитів (Query Optimizer), який оцінює різні шляхи виконання запиту та вибирає найкращий план виконання (execution plan) за найменшою ціною виконання; виконувач запитів (Query Executor), який послідовно діє за створеним планом виконання, взаємодіючи при цьому з менеджером сховища.

***Менеджер сховища*** (Storage Engine) включає: модуль, що відповідає за методи доступу до бази даних (Access Methods), який обробляє всі запити вводу/виводу (handles I/O) до рядків (rows), індексів (indexes), сторінок (pages), розміщень (allocations) та версій рядків (row versions); менеджер буферу (Buffer Manager), який управляє виділенням оперативної пам'яті буферу пулу (buffer pool); менеджер транзакцій (Transaction Manager), який відповідає за блокування даних (handles the locking of data) для забезпечення ізоляції ACID-транзакції та управляє логом транзакцій (transaction log).

*Пул буферів* (The Buffer Pool) – основний споживач оперативної пам'яті в SQL Server. Він включає кеш-сховища, основними з яких являється кеш плану та кеш даних.

*Мережевий інтерфейс* (SQL Server Network Interface – SNI) – рівень протоколів, який відповідає за здійснення клієнт-серверної взаємодії. Включає набор APIs, які використовуються системою управління базами даних (database engine) та клієнтським драйвером SQL Server Native Client (SNAC).

Мережевий інтерфейс підтримує наступні протоколи.

*Shared memory.* Використовується для підключення клієнтських додатків, що виконуються локально (розташовані на тому же комп'ютері, що і сервер).

*TCP/IP.* Основний протокол віддаленого доступу. Дозволяє віддаленим додаткам-клієнтам підключитися до серверу баз даних з використанням IP-адреси та номеру порту (порти за замовчанням TCP – 1433, UDP – 1434).

*Named Pipes.* Протокол Named Pipes був розроблений для локальних мереж (LANs) та може бути неефективним для глобальних мереж. Named Pipes використовує порт TCP 445.

*VIA (Virtual Interface Adapter)* – протокол для здійснення високопродуктивної взаємодії між двома системами. Такий протокол вимагає встановлення спеціальних фізичних VIA-пристроїв, які підтримують взаємодію. Прикладом таких пристроїв являються мережі сховищ даних (SAN storage area networks).

*Точки доступу TDS (tabular data stream – потік табличних даних)* – протокол Microsoft, який використовується для взаємодії з сервером баз даних. Після здійснення підключення клієнтського додатку до сервісу SNI створюється захищене з'єднання з точкою доступу TDS, яка виконує роль посередника у взаємодії клієнта та сервера. З кожним мережевим протоколом пов'язані дві точки доступу TDS: одна – звичайна, друга – додаткова для доступу адміністратора (DAC – dedicated administrator connection). Так, наприклад, SQL-команда SELECT пересилається як TDS повідомлення через канал, забезпечений з'єднанням по протоколу TCP/IP. Рівень протоколу відповідає за розпакування повідомлень, що надходять. Таким чином SQL-команда SELECT, яка позначена у TDS-пакеті як повідомлення типу "SQL Command", буде перенаправлена на компонент синтаксичного аналізатора запитів Query Parser і далі на оптимізацію та виконання.

### 4.2.1.Життєвий цикл команд SELECT та UPDATE

Розглянемо ***життєвий цикл команди SELECT***.

1. Компонент SQL Server Network Interface (SNI), з яким взаємодіє клієнтський додаток, створює підключення до SNI серверу баз даних SQL Server, використовуючи для цього протокол TCP/IP. Підключення автоматично створюється до точки доступу TDS. Клієнт пересилає команду SELECT на сервер SQL Server як TDS повідомлення.

2. SNI сервера розпаковує TDS повідомлення, зчитує команду SELECT та перенаправляє її до синтаксичного аналізатора (Command Parser).

3. Command Parser здійснює пошук готового плану, який відповідає даній команді в кеш-буфері планів. Якщо готового плану не знайдено, він створює дерево запитів та передає його до модулю оптимізатора для створення плану виконання.

4. У разі простого запиту оптимізатор створює тривіальний план (zero cost or trivial plan) та передає його на модуль виконання (Query Executor).

5. В момент виконання модуль Query Executor визначає, які дані потрібно зчитати з бази даних для реалізації плану та посилає запит до модулю методів доступу до бази даних (Access Methods), використовуючи при цьому інтерфейс OLE DB.

6. Модуль Access Methods надсилає запит на отримання необхідної сторінки бази даних до модулю Buffer Manager.

7. Модуль Buffer Manager перевіряє наявність сторінки у кеші даних і у разі відсутності зчитує сторінку з диску, розташовує її в кеші та повертає модулю Access Methods.

8. Модуль Access Methods повертає отриманий результат до обробника запитів, який у свою чергу надсилає їх кліенту.

Розглянемо особливості ***життєвого циклу команди UPDATE***. До моменту отримання запиту модулем доступу до даних (Access Methods) шлях цієї команди повторює шлях команди запиту (шаг 8). Далі *модуль доступу*

(Access Methods) звертається до компоненту *менеджера транзакцій* (Transaction Manager), який включає два важливих компонента: менеджер блокувань (Lock Manager) та менеджер лога (Log Manager). Lock Manager відповідає за забезпечення конкурентного доступу до даних відповідно до рівню ізоляції шляхом застосування механізму блокувань (locks).

*Log Manager* відповідає за фіксацію в файлі логу транзакцій змін, які створюються модулем Access Methods. Такий тип ведення логу називається Write-Ahead Logging. На основі логу транзакцій та запису робочих змін на диск створюється можливість відновлення роботи (recovery) SQL Server після системного збою без значних втрат. В лог транзакцій записуються деталі змін в сторінках, які були створені в результаті обробки-модифікації (для кожної команди INSERT, DELETE, UPDATE, BEGIN TRANSACTION, END TRANSACTION, COMMIT, ROLLBACK, CHECKPOINT). Запис до логу (log entry) заноситься перед реальною операцією зміни.

Таким чином, зміни, які були отримані в результаті виконання команди UPDATE будуть зафіксовані і актуалізація модифікації буде здійснена тільки у тому випадку, коли буде отримане підтвердження, що операція була записана в лог транзакцій. У разі отримання підтвердження від менеджера транзакцій, модуль Access Methods передає запит на модифікацію до модулю менеджера буферу Buffer Manager, завершаючи зміну даних.

*Buffer Manager*. Якщо необхідна сторінка вже існує в кеш-пам'яті – менеджер буферу модифікує її відповідно до запиту модулю Access Methods та повертає підтвердження, що модифікація зроблена. Важливим моментом являється те, що команда UPDATE змінює дані тільки в буфері (кеші-даних) та не вносить змін до актуальної бази даних (файлу, який розташований на диску). Причина такого підходу полягає в необхідності покращення продуктивності системи (аналогія механізму кеш-памяті в комп'ютерних системах). Така сторінка називається брудною сторінкою (dirty page) тому, що вона не відповідає актуальній інформації бази даних. Введення цього механізму не порушує принципу надійності змін (durability). Так, у разі вимкнення жив-

лення, за рахунок логу транзакцій можливо створити відновлення всіх змін. Якщо необхідної сторінки не існує в пам'яті – вона завантажується з оперативної пам'яті до кешу та позначається як чиста сторінка (clean page), тобто така, що відповідає актуальній інформації. Брудні сторінки актуалізуються (записуються на диск) періодично або по досягненні контрольної точки (checkpoint). Вільний простір у буфері введений для швидкого розташування нових сторінок і у разі його зменшення нижче встановленого обмеження (threshold) старі сторінки записуються на диск, актуалізуючи зміни та звільнюючи буфер. За підтримку вільного простору у буфері та актуалізацію змін відповідає сервіс відкладеного запису (lazywriter). Цей сервіс також відповідає за моніторинг та управління фізичною пам'яттю, яка використовується сервером – наприклад здійснює підключення додаткової пам'яті в залежності від конфігурації.

Інший механізм базується на застосуванні контрольних точок (checkpoint process). Контрольні точки визначають моменти часу, в які всі модифіковані брудні сторінки повинні бути актуалізовані (записані на диск). Таким чином контрольна точка стає маркером, відносно якого може здійснюватися процедура відновлення (recovery). Актуалізовані сторінки не видаляються з буферу, а лише помічаються як чисті. В SQL Server інтервал між контрольними точками дорівнює однієї хвилини, але актуалізація даних здійснюється лише у разі запису розміром 10MB в лог-файл за цей інтервал. Процедура управління контрольними точками може викликатися користувачем (в запиті або в збереженій процедурі) з використанням команди T-SQL CHECKPOINT.

### 4.2.2. Рівні ізоляції в MSSQL Server.

Для забезпечення безпеки та надійності обробки транзакцій SQL Server застосовує декілька рівнів ізоляції. Чим нижче рівень ізольованості транзакцій застосується у роботі, тим більший ризик виникнення проблем, але тим вище й швидкість паралельної обробки запитів користувача. Для подальшого

розгляду особливостей рівнів ізольованості в SQLServer розглянемо три додаткові **проблеми (феномена) залежності транзакцій** [Bolton, 2010]: втрачені оновлення (lost updates), повторного зчитування (double reads), ефект Helloween.

**Втрачені оновлення (Lost Updates).**

Проблема виникає у тому випадку коли система не відстежує залежності між транзакціями та дозволяє актуалізацію змін даних в області, яка в момент зміни обробляється (з метою модифікації) іншою транзакцію.

Наприклад, при обробці конкурентних запитів на оновлення даних маємо:

| ***Transaction 1*** | ***Transaction 2*** |
|---|---|
| ** Запит 1 ** <br> **SELECT** * **FROM** users <br> **WHERE** age **BETWEEN** 10 **AND** 30; | |
| | ** Запит 2 ** <br> **UPDATE** users **SET** salary=salary + 1000 <br> **WHERE id = 1**; <br> **COMMIT;** |
| **WAITFOR DELAY '00:00:08.000';** | |
| ** Запит 1 ** <br> **UPDATE** users **SET** salary = salary + 10 <br> **WHERE id = 1**; <br> **COMMIT;** | |

В результаті паралельної обробки таких транзакцій отримуємо підвищення доходу співробітника лише на 10 одиниць замість 1010. Для моделювання ситуації можна використовувати команду WAITFOR DELAY '00:00:08.000' в тілі першої транзакції. А команда SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED дозволяє відповідний рівень ізольованості транзакції.

Приклад процедур, що моделюють цю ситуацію, має наступний вигляд:

```
/* SESSION 1*/
USE AdventureWorks;
```

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
DECLARE @SafetyStockLevel int = 0, @Uplift int = 5;
BEGIN TRAN;
/*Отримання рівня запасів*/
SELECT @SafetyStockLevel = SafetyStockLevel FROM Product WHERE ProductID = 1;
/*Оновлюємо значення змінної*/
SET @SafetyStockLevel = @SafetyStockLevel + @Uplift;
WAITFOR DELAY '00:00:05.000';
/*Оновлення рівня запасів для продукту з кодом 1*/
UPDATE Product SET SafetyStockLevel = @SafetyStockLevel WHERE ProductID = 1;
SELECT SafetyStockLevel FROM Product WHERE ProductID = 1;
COMMIT TRAN;

/* SESSION 2*/
USE AdventureWorks;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
DECLARE @SafetyStockLevel int = 0,@Uplift int = 100;
BEGIN TRAN;
/*Отримання рівню запасів*/
SELECT @SafetyStockLevel = SafetyStockLevel FROM Product WHERE ProductID = 1;
/*Оновлюємо значення змінної*/
SET @SafetyStockLevel = @SafetyStockLevel + @Uplift;
/*Оновлення рівня запасів для продукту з кодом 1*/
UPDATE Product SET SafetyStockLevel = @SafetyStockLevel WHERE ProductID = 1;
SELECT SafetyStockLevel FROM Product WHERE ProductID = 1;
COMMIT TRAN;
```

Структура таблиці Product тривіальна. В результаті отримуємо ситуацію, в якій значення рівня запасів (SafetyStockLevel) для продукта з кодом 1 дорівнює 1005, при актуальному значенні поля SafetyStockLevel 1000 до запуску транзакцій.

**Повторне зчитування (Double Reads).**

Проблема повторного зчитування виникає у разі застосування блокування на рівні рядків та паралельної обробки області даних двома транзакці-

ями – у той час коли одна транзакція здійснила обробку рядка, він був переставлений другою транзакцією таким чином, що він буде повторно зчитаним першою транзакцією (рис. 15). Розглянемо приклад в якому рядок з даними по Bethany Raheem буде зчитаним двічі. В базі даних Adventureworks існує 5 чоловік з прізвищем Raheem. В першій сесії здійснюється оновлення даних, вона блокує рядок всередині області, що містить дані, які пов'язані з прізвищем Raheem. Друга сесія сканує всі рядки, в яких значення поля LastName починається з Raheem. Цей запит сканує індексний файл, та вибірка блокується командою оновлення транзакції 1.

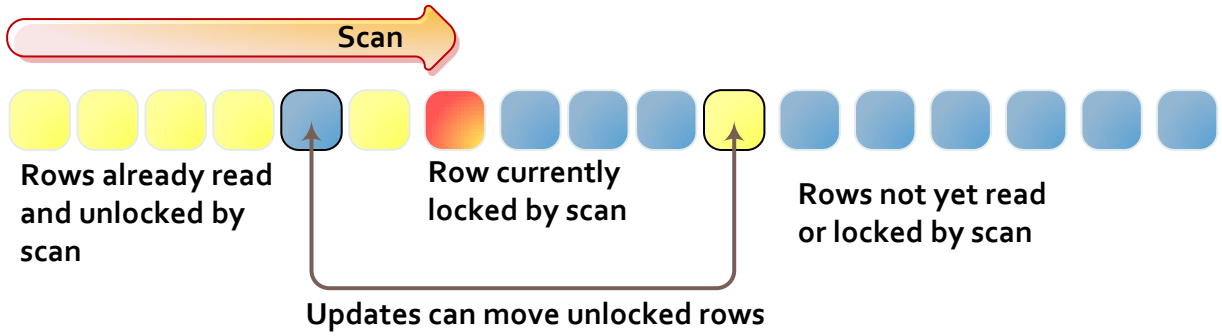



Рис.15. Пояснення проблеми повторного зчитування

| ***Transaction 1*** | ***Transaction 2*** |
|---|---|
| */* Запит 1 */*<br>**USE** AdventureWorks;<br>**BEGIN TRAN**<br>**UPDATE** Person.Person **SET** LastName = 'Raheem_DOUBLE_READ_BLOCK'<br>**WHERE** LastName = 'Raheem'<br>**AND** FirstName = 'Kurt'; | |
| | */* Запит 2 */*<br>**USE** AdventureWorks2008;<br>**SELECT** FirstName,LastName<br>**FROM** Person.Person<br>**WHERE** LastName Like 'Raheem%'; |
| **WAITFOR DELAY '00:00:08.000';** | |
| */* Запит 3 */*<br>**UPDATE** Person.Person<br>**SET** LastName = 'Raheem_DOUBLE_READ_REAL'<br>**WHERE** LastName = 'Raheem'<br>**AND** FirstName = 'Bethany';<br>**COMMIT TRAN;** | |

Після оновлення заблокованого рядку, здійснюється читання незаблокованих даних командою вибірки транзакції 2, але одночасно з цим читанням командою оновлення транзакції 1 здійснюється блокування та оновлення вже вибраного в транзакції 2 рядку. При цьому оновлення змінює порядок цього рядку у індексному файлі (за яким проводиться сканування таблиці транзакцією 2), переставляючи його в місце, до якого не дійшла процедура сканування транзакції 2 – транзакція 2 повертає два варіанта рядку – старий, зчитаний до оновлення, та новий.

**Ефект Halloween.**

Ефект Halloween пов'язаний зі сценарієм, який обумовлений політикою обробки конкурентних запитів, за яким дані в момент виконання процедури оновлення можуть переміщуватися за рахунок зміни індексу. Ситуація схожа на проблему подвійних зчитувань, але пов'язана також і з оновленням даних.

Для оновлення набору рядків командою UPDATE перш за все виконується зчитування: при реалізації застосовуються два курсори (cursor) – один для читання, другий для запису. Якщо рядок був оновлений курсором запису і це оновлення впливає на позицію рядку (ситуація подвійного зчитування), даний рядок може бути зчитаний та оновлений другий раз.

В SQL Server існує надійний механізм захисту проти виникнення цієї ситуації – ідея захисту полягає в повному зчитуванні області рядків, які обробляються командою UPDATE (це указується в плані виконання команди) до тимчасової бази даних (tempdb), перед завантаженням їх у курсор. Для підвищення ефективності існують механізми виявлення можливості виникнення такого ефекту.

### 4.2.3. Блокування в SQLServer

Для кращого розуміння сутності песимістичної та оптимістичної конкурентної обробки транзакцій слід розглянути механізми блокування, які застосовуються в SQLServer. ***Блокування*** – форма диспетчеризації управління ресурсом (resource scheduling). Режим блокування (mode of the lock) визначає

можливість використання ресурсу паралельно різними завданнями (процедурами). В системі існує єдине динамічне представлення (dynamic management view – DMV), яке називається Sys.dm_tran_locks, воно використовується для аналізу блокувань. Атрибути цього представлення можуть бути поділеними на дві групи: характеристики ресурсу, на якому існує запит (таблиця 4), характеристики обробки цього запиту (таблиця 5).

Основні характеристики ресурсу

Таблиця 4

| І'мя колонки | Описання |
|---|---|
| Resource_type | Описує тип ресурсу OBJECT, PAGE, KEY (див. табл.4.) |
| Resource_subtype | Підтип ресурсу. Наприклад, у разі створення таблиці у транзакції в цьому полі буде указаний підтип DDL при вказаному типі DATABASE. |
| Resource_database_id | Посилання на базу даних, до якої відноситься блокуємий ресурс. |
| Resource_description | Додаткова інформація про ресурс. |
| Resource_associated_ entity_id | Описує сутність (Object ID, HoBT ID, Allocation Unit ID), з якою пов'язане блокування. |

Характеристики ресурсу

Таблиця 5

| Ім'я колонки | Описання |
|---|---|
| Request_mode | Режим блокування. Наприклад, IX (Intent Exclusive), X (Exclusive) та ін. |
| Request_status | Статус запиту: GRANT – наданий даному запиту; WAIT – очікування можливості; CONVERT – знаходиться у стадії перетворення до більш жорсткого режиму блокування. |
| Request_reference_count | Кількість раз блокування даного ресурсу. |
| Request_session_id | Сесія в якій здійснюється запит ресурсу. Може мати спеціальні значення: 2 - у разі осиротілої розподіленої транзакції (an orphaned distributed transaction), 3: відстроченої транзакції відновлення (deferred recovery |

| | |
|---|---|
| | transaction). |
| Request_exec_context_id | Контекст процесу обробки |
| Request_request_id | Batch ID запиту, що володіє ресурсом |
| Request_owner_type | Тип сутності власника запиту. Наприклад: TRANSACTION, CURSOR, SESSION, SHARED_TRANSACTION_WORKSPACE, EXCLUSIVE_TRANSACTION_WORKSPACE |
| Request_owner_id | Використовується у разі виконання транзакції – тобто якщо значення поля Request_owner_type – Transaction, та містить transaction ID |
| Request_owner_guid | Використовується у разі виконання розподіленої транзакції. Значення дорівнює ідентифікатору MSDTC GUID транзакції. |

Гранулярність блокування (Lock Granularity).

В SQL Server впроваджено багато варіантів гранулярності блокування, які пов'язані з різноманітними сценаріями обробки. Коли процес виконання транзакції взаємодіє з даними, як правило застосовуються декілька блокувань різноманітних ресурсів. Висока модульність означає більший рівень паралелізації обробки. В таблиці 6 наведені основні типи ресурсів

Типи ресурсів

Таблиця 6

| Рівень | Приклад | Пояснення |
|---|---|---|
| RID | 1:8185:4 | Для блокування рідка використовується ідентифікатор рядка (Row ID). Здійснюється у разі використання таблиці як купи (heap), тобто без індексу. Формат: <FILE : PAGE : SLOT ID> |
| KEY | (3a01180ac47a) | Блокування рядку на основі індексу. Здійснюється у разі використання в запиті таблиці з кластерним індексом. |
| PAGE | 1:19216 | Блокування сторінки віртуальної пам'яті, яка містить дані або індекс. Формат: <FILE ID>:<PAGE NUMBER>, що відображаються на поля fi_le_id та page_id в таблиці |

| | | |
|---|---|---|
| | | sys.dm_os_buffer_descriptors. |
| EXTENT | 1:19216 | Блокування набору з 8 сторінок. <FILE ID> : <FIRST PAGE NO> |
| HoBT | 7205759405863 7312 | Блокування HoBT (Heap or Balanced Tree) купи або збалансованого дерева. Якщо таблиця без індексу – блокується купа (heap), у разі індексу – блокується збалансоване дерево індексу (BTree of the index). |
| OBJECT | 2105058535 | Як правило, означає блокування таблиці. При цьому здійснюється блокування всіх сторінок з даними та індексами. |
| APPLICATION | 0:[MyAppLock]: (6731eaf3) | Блокування додатку. Встановлюється sp_getapplock. |
| METADATA | xml_collection_ id = 65536 | Блокування метаданих. |
| ALLOC- ?TION_UNIT | 7205759403982 8480 | Allocation Unit ID використовується у разі виконання відстрочених операцій – видалення великих таблиць, операцій типу SELECT INTO. |
| FILE | 0 | Блокування на рівні файлів. Використовуються у разі додавання нових файлів до бази даних. |
| DATABASE | 7 | Блокування на рівні бази даних. Приклади: зміна структури бази даних, зміна режимів доступу з read_write на read_only. |

### 4.2.4. Ієрархія рівнів блокування

SQL Server має декілька рівнів блокування, які відрізняються модульністю (lock at varying degrees of granularity) – це рівень об’єктів, сторінок, рядків. На верхньому рівні ієрархії знаходиться таблиця, з якою пов’язана певна кількість сторінок, що відносяться до трьох базових типів одиниць розподілу пам’яті (allocation units – 4 КВ, 8 КВ, 64 КВ). Кожна сторінка містить певну кількість рядків (rows). У разі запита на оновлення даних проводиться пошук існування ексклюзивного блокування запитуємої області даних (рядка, сторінки, таблиці). Так, наприклад, у разі оновлення структури таблиці

сервіс менеджера блокувань (lock manager) здійснює пошук блокування на відповідному рівні. Рівнем блокування для виконання команди ALTER TABLE являється блокування схеми таблиці (schema modification lock) – це означає, що дана команда не може виконуватися одночасно з оновленням рядка цієї таблиці. Питання сумісності блокувань (lock compatibility) виходить за рамки цієї роботи, деталі доступні на сайті http://msdn.microsoft.com/en-us/library/ms186396.aspx.

Важливою проблемою є знаходження балансу між швидкістю обробки та збереженням характеристики ізольованості транзакції. Для здійснення цього балансування використовується рівень ізольованості за умовчанням для MSSQL серверу — *read committed.*

Під час виконання запиту на цьому рівні ізольованості виконуються короткотривалі блокування рядка за рядком (read locks are only taken for the duration of the read, not for the duration of the transaction). Тривалість спільних блокувань просто занадто довга для того, щоб зчитати та обробити кожен рядок; сервер зазвичай звільняє кожне блокування перед тим, як перейти до наступного рядка. Тому, якщо запустити на виконання простий вираз *select* у режимі *read committed* та перевірити на існування блокувань (напр. s-sys.dm_tran_locks), ми одержимо єдине блокування рядка. Основна мета цих блокувань — пересвідчитися, що цей вираз тільки зчитує і повертає підтверджені дані. Блокування працюють тому, що оновлення завжди вимагають виняткове блокування, яке блокує будь-які зчитування, що намагаються звернутися до спільного блокування.

Оскільки сканування блокує тільки один рядок за раз, ніщо не заважає паралельному оновленню перенести рядок до чи після того, як процедура сканування досягне його. Але при цьому виникає проблема подвійних зчитувань рядків (double reads). Для вирішення цієї проблеми використовується механізм повторного зчитування (repeatable read).

На відміну від обробки запитів у режимі ізоляції read committed, процедура повторного зчітування repeatable read зберігає блокування на кожному

рядку (рис.16). Навіть рядки, які не відносяться до запиту, залишаються заблокованими. Такий тип блокування забезпечує те, що рядки, захоплені запитом, не можуть бути змінені чи видалені паралельною сесією, поки поточна транзакція не завершиться. Але блокування не захищають рядки, які ще не були просканові, від змін, видалень та не запобігають вставці нових рядків в область, яка обробляється.

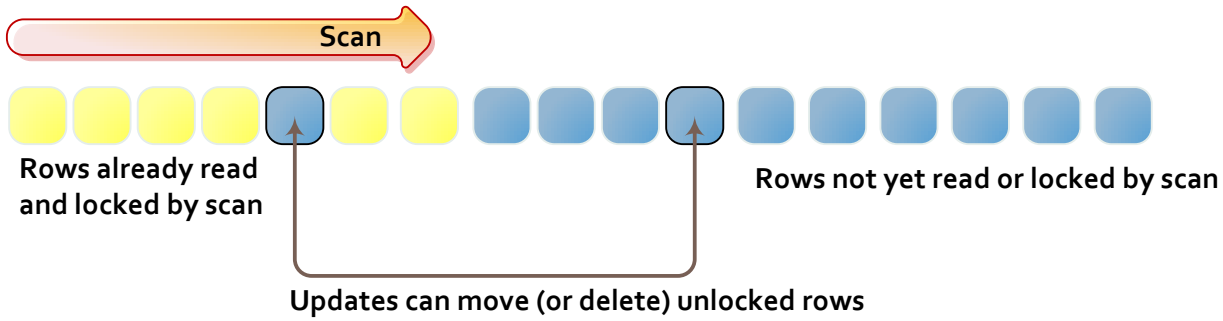



Рис.16. Пояснення проблеми повторного зчитування

Таким чином існує можливість вставити новий “фантомний” рядок поміж заблокованих рядків, що вже були просканові. Розглянемо декілька прикладів того, як ми можемо отримати неочікувані результати на рівні ізольованості *repeatable read*.

Переміщення рядків. Для експерименту потрібні дві сесії та таблиця, код ініціювання якої показаний нижче.

```
CREATE TABLE t (a int PRIMARY KEY, b int);
INSERT t VALUES (1, 1);
INSERT t VALUES (2, 2);
INSERT t VALUES (3, 3);
```

| *Transaction 1* | *Transaction 2* |
|---|---|
| *-- Запит 1* | |
| **BEGIN TRAN**<br>**UPDATE** t **SET** b = 2 **WHERE** a = 2; | |
| **WAITFOR DELAY ‘00:00:08.000’;** | *-- Запит 2*<br>**SELECT** * **FROM** t **WITH** (repeatableread); |

У сесії 1 (transation 1) першою командою здійснюється блокування другого рядка. Далі у сесії 2 командою вибірки запускається сканування таблиці в режимі *repeatable read*. Це сканування зчитує перший рядок, потім блокується, очікуючи, поки сесія 1 не звільнить блокування другого рядка. Поки сканування заблоковане, у сесії 1 здійснюється переміщення третього рядка на початок таблиці. Підтвердження транзакції 1 звільнює блокування сесії 2, яка продовжує роботу, тобто повністю пропускає третій рядок і повертає лише два перші рядки.

Слід також відзначити, що якщо експеримент буде змінено так, щоб сесія 1 намагалася зачепити перший рядок у таблиці, це спричинить затор у сесії 2, яка утримує блокування на цьому рядку.

Фантомні рядки. Розглянемо ситуацію в якій фантомні рядки можуть викликати неочікувані результати. Для цього створені дві таблиці.

```
CREATE TABLE t 1 (a1 int PRIMARY KEY, b1 int);
INSERT t1 VALUES (1, 9);
INSERT t1 VALUES (2, 9);
CREATE TABLE t 2 (a2 int PRIMARY KEY, b2 int);
```

Виконуються дві транзакції

| ***Transaction 1*** | ***Transaction 2*** |
|---|---|
| ** Запит 1 ** | |
| **BEGIN TRAN**<br>**UPDATE** t1 **SET** a1 = 2 **WHERE** a1 = 2; | |
| **WAITFOR DELAY '00:00:08.000';** | ** Запит 2 ** |
| | **SELECT** * **FROM** t1 **LEFT OUTER JOIN** t2 **ON** b1=a2; |
| ** Запит 3 ** | |
| **INSERT** t2 **VALUES** (9, 0);<br>**COMMIT TRAN** | |

Розглянемо план виконання команди

**SELECT** * **FROM** t1 **LEFT OUTER JOIN** t2 **ON** b1=a2,

який можна отримати у текстовому вигляді застосовуючи команду ***set statistics profile on.***

```
  |--Nested Loops(Left Outer Join,
WHERE:([dbtest].[dbo].[t1].[b1]=[dbtest].[dbo].[t2].[a2]))

       |--Clustered Index
       Scan(OBJECT:([dbtest].[dbo].[t1].[PK__t1__3213A9FA214C51F7]))

       |--Clustered Index
       Scan(OBJECT:([dbtest].[dbo].[t2].[PK__t2__3213A9FBD12D6581]))
```

Таким чином команда об'єднання використовує об'єднання вкладених циклів. Тобто алгоритм сканує перший рядок із t1, намагається об'єднати його з t2, знаходить, що немає жодних співпадаючих рядків і виводить нульовий рядок. Далі він блокується, очікуючи, поки сесія 1 не звільнить блокування на другому рядку із t1. Далі, сесія 1 вставляє новий рядок у t2 і звільняє блокування. У результаті отримуємо

| a1 | b1 | a2 | b2 |
|---|---|---|---|
| 1 | 9 | NULL | NULL |
| 2 | 9 | 9 | 0 |

*Read uncommitted* ідентифікує найнижчий рівень ізольованості. Він дозволяє зчитувати змінені дані непідтверджених трансакцій (dirty reads). Розглянемо наступний приклад:

```
CREATE TABLE t 1 (a1 int, b1 int);
INSERT t1 VALUES (0, 0);  INSERT t1 VALUES (1, 1);
CREATE TABLE t 2 (a2 int PRIMARY KEY);
INSERT t2 VALUES (0);  INSERT t2 VALUES (1)
```

| ***Transaction 1*** | ***Transaction 2*** |
|---|---|
| */* Запит 1 */* | |
| **BEGIN TRAN**<br>**UPDATE** t2 **SET** a2 = a2 **WHERE** a2 = 0; | |
| | */* Запит 2 */* |
| **WAITFOR DELAY '00:00:08.000';** | **SELECT** * **FROM** t1 **WITH** (nolock)<br>**WHERE EXISTS (SELECT** * **FROM** t2<br>**WHERE** t1.a1 = t2.a2**);** |
| */* Запит 3 */* | |
| **DELETE FROM** t1 **WHERE** a1=0;<br>**COMMIT TRAN** | |

У сесії 1 блокується перший рядок таблиці t2. Далі, у сесії 2 запускається запит, який використовує наступну схему

**SELECT** * **FROM** t1 **WITH** (nolock) **WHERE EXISTS (SELECT** * **FROM** t2 **WHERE** t1.a1 = t2.a2**);**

```
|--Nested Loops(Left Semi Join,
WHERE:([dbtest].[dbo].[t1].[a1]=[dbtest].[dbo].[t2].[a2]))
     |--Table Scan(OBJECT:([dbtest].[dbo].[t1]))
        |--Clustered Index
        Scan(OBJECT:([dbtest].[dbo].[t2].[PK__t2__3213A9FB0649E672]))
```

Процедура сканування таблиці отримує перший рядок таблиці t1 без ініціювання будь-яких блокувань і потім намагається об'єднати цей рядок з t2. Оскільки перший рядок t2 заблокований і сканування, що використовує кластерний індекс, запускається за умовчанням на рівні read committed, запит блокується. Нарешті, у сесії 1 видаляється перший рядок t1 та підтверджується завершення транзакції. Тепер запит у сесії 2 може продовжитися. Однак, поки він був заблокований, ми видалили рядок, який він намагається об'єднати. Запит намагається зчитати більше даних з видаленого рядка і завершається невдачею із наступною помилкою «Could not continue scan with NOLOCK due to data movement». Таким чином сканування у режимі *read uncommitted* чи *nolock* може не тільки спричинити неочікувані результати, а навіть може викликати повну відмову запиту.

Слід відзначити, що у разі створення таблиці з кластерним індексом означена вище помилка відсутня, але результат також не вірний – повертається два рядка, тобто видалення рядка не враховується. У разі створення таблиці командою **CREATE TABLE** t1 (a1 int **PRIMARY KEY**, b1 int) маємо наступний план виконання

**SELECT** * **FROM** t1 **WITH** (nolock) **WHERE EXISTS (SELECT** * **FROM** t2 **WHERE** t1.a1 = t2.a2**);**

```
  |--Nested Loops(Left Semi Join,
  WHERE:([dbtest].[dbo].[t1].[a1]=[dbtest].[dbo].[t2].[a2]))
       |--Clustered Index
       Scan(OBJECT:([dbtest].[dbo].[t1].[PK__t1__3213A9FAB6621FDD]))
       |--Clustered Index
       Scan(OBJECT:([dbtest].[dbo].[t2].[PK__t2__3213A9FBB30BAED9]))
```

Зведена таблиця 7 показує зв'язок проблем та рівнів ізольованості в SQLServer. Маркером «+» помічені проблеми, які вирішуються на даному рівні ізольованості.

Зв'язок проблем та рівнів ізольованості в SQLServer.

Таблиця 7

| Рівень ізольованості | Проблеми | | | | | |
|---|---|---|---|---|---|---|
| | Halloween | Dirty reads | Lost update | Double read | Non-repeatable reads | Phantoms |
| Read Uncommitted | + | | | | | |
| Read Committed | + | + | | | | |
| Repeatable Read | + | + | + | + | + | |
| Serializable | + | + | + | + | + | + |

### 4.3.Особливості обробки розподілених транзакцій в системі Windows

Бібліотеки ADO.NET (ActiveX Data Objects .NET) пропонують модель програмування із явним управлінням транзакціями. Розробник відповідає за явний запуск та управління транзакцією. Розглянемо приклад.

```
string connectionString = "...";
IDbConnection connection = new SqlConnection(connectionString);
connection.Open();
IDbCommand command = new SqlCommand();
command.Connection = connection;
IDbTransaction transaction;
transaction = connection.BeginTransaction(); //Enlisting database
```

```
command.Transaction = transaction;
try
{
        /* Логіка взаємодії з сервером */
        transaction.Commit();
}
catch
{
        transaction.Rollback(); //Abort transaction
}
finally
{
        connection.Close();
}
```

Виклик методу BeginTransaction() повертає реалізацію інтерфейсу IDbTransaction, що відповідає за управління транзакцією. Якщо усі оновлення чи інші зміни, зроблені над БД, не викликали помилок – викликається метод успішного завершення підтвердження Commit(). Якщо виникла помилка – транзакція відміняється методом Rollback().

Оскільки явна модель програмування проста і не вимагає класу, виконуючого транзакцію, вона найбільш придатна для взаємодії одного об’єкта з однією базою даних (або одним транзакційним ресурсом).

Проблема виникає у тому разі, коли об’єкти починають взаємодію в рамках транзакції між собою, маючи при цьому доступ до однієї бази (рис.17). Об’єкти можуть бути розподілені поміж різних машин, і проблеми, що пов’язані зі стійкою роботою мережі та надійністю машин-вузлів додають складність для управління транзакцією.

Можливим рішенням являється включення логіки взаємодії для координації транзакцій в об’єктах, але такий підхід не гнучкий – незначні зміни бізнес логіки, кількості залучених об’єктів вимагають значних витрат, пов’язаних з переробкою системи. Можлива гетерогенність об’єктів також додає складності забезпечення та підтримки такої координації.

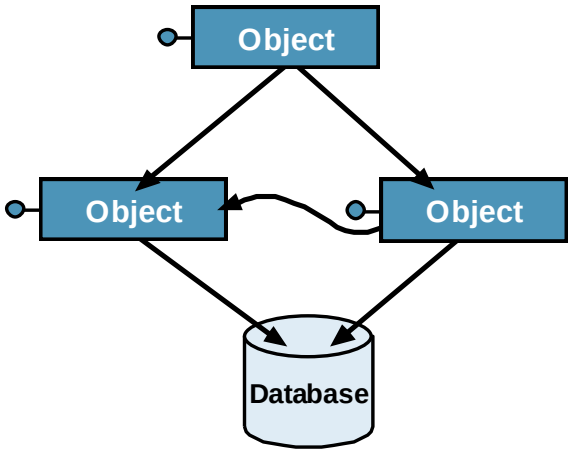



Рис. 17. Доступ багатьох об’єктів до одного ресурсу

Ситуація стає більш складною, коли в транзакції приймає участь багато ресурсів, тобто у разі виконання розподіленої транзакції (рис. 18). Розподілена транзакція складається із двох чи більше незалежних партій об’єктів (часто у різних виконавчих середовищах), або двох чи більше транзакційних ресурсів, або і багатьох об’єктів та багатьох ресурсів. Кожен із ресурсів може відмовити або підтвердити свою частину розподіленої транзакції. Намагатися явно керувати усіма потенційними випадками відказу розподіленої транзакції являється неефективним. Для розподіленої транзакції, як це було описано вище, необхідно покладатися на двох-етапний протокол підтвердження та на ряд спеціальних програмних засобів (сервісів), основу яких створює менеджер розподілених транзакцій. Таким менеджером у продуктах Windows являється координатор розподілених трансакцій.

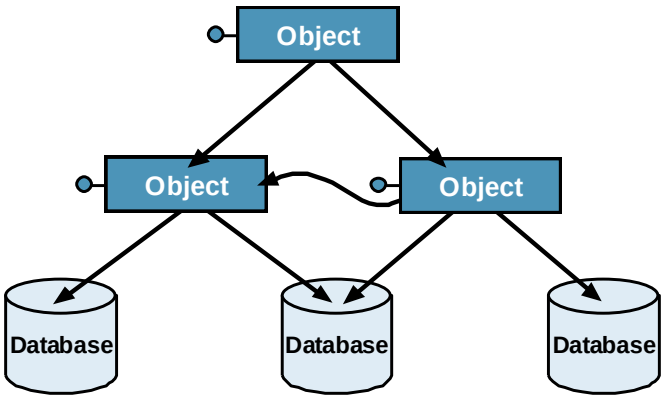



Рис. 18. Доступ багатьох об’єктів до багатьох ресурсів

### 4.3.1. Служба координатора розподілених транзакцій

Служба координатора розподілених транзакцій (Distributed transaction coordinator – далі DTC) встановлюється на кожній із систем Microsoft Windows 2000 Server, Windows XP, Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, Windows Server 2008 R2, Windows 8, Windows Server 2012. Кожний з вказаних класів продуктів має свій власний зразок DTC. Слід відзначити, що DTC вперше був випущений як частина Microsoft SQL Server версії 6.5 та включений у Microsoft Transaction Server (MTS), для забезпечення низькорівневої інфраструктури для транзакцій.

У деяких випадках, транзакції можуть бути більш ефективними, коли дві чи більше системи налаштовані на роботу з єдиним менеджером транзакцій DTC. Наприклад, на серверному кластері, єдиний сервіс DTC обслуговує увесь кластер. При цьому у разі розподіленої транзакції, яка обробляється багатьма вузлами, менеджер транзакцій використовує тільки один лог ( журнал фіксуючий зміни).

Якщо додаток виконує транзакцію, залучаючи до цього два чи більше комп'ютери, слід засвідчитися в можливості їх мережної взаємодії. DTC може поширювати транзакцію від однієї системи до іншої, тільки якщо процес DTC на першій чи первинній системі може взаємодіяти із процесом DTC на вторинній системі. Проблема мережної конфігурації на тій чи іншій системі може перешкодити взаємодії процесів DTC, спричиняючи помилку «не вдалося заручитися» (failed to enlist).

Взаємодія (process-to-process comunication) між процесами DTC базується на сервісах, які забезпечують взаємодію віддалених додатків по протоколам Local Remote Procedure Calls (LRPC), Remote Procedure Calls (RPC), чи Transaction Internet Protocol (TIP). DTC використовує наступні системні сервіси:

- NTLMSSP ( NT LAN Manager Security Support Provider) – протокол передачі бінарних даних, який використовується функціями SSPI (Security Support Provider Interface – API для здійснення операцій, пов'язаних з безпекою) для

здійснення операцій безпечного з'єднання та взаємодії. В DTC сервіс використовується для шифрування строки підключення (open string) до бази даних XA (бази, що підтримують протокол обробки розподілених транзакцій XA специфікації Open Group), який він зберігає у лог-файлі. Відкритий рядок шифрується для того, щоб захистити ідентифікатори та паролі будь-якого користувача, які є у відкритому рядку.

• RPCSS (Remote Procedure Call Services ) для взаємодії між компонентами DTC.

Розглянемо особливості обробки розподілених транзакцій сервісом DTC. Кожен з вузлів системи має локальний менеджер транзакцій. Усі додатки та менеджери ресурсів взаємодіють із своїми локальними менеджерами транзакцій. *Менеджером ресурсів* називається системний сервіс, що управляє даними, які зберігаються на диску (в англомовних джерелах використовується термін persistent data, що означає – стійки дані), на відміну від оперативних даних. Менеджери транзакцій спільно управляють системою.

Коли приходить запит із новою транзакцією, між двома системами встановлюється зв'язок начальник-підлеглий (superior-subordinate relat-?onship). Менеджер транзакцій системи, що робить запит, стає начальником над менеджером транзакцій системи, що приймає запит. Система начальник управляє усіма підлеглими системами використовуючи двохетапний протокол підтвердження. Кожна система може мати багато підлеглих систем, але вона може мати максимум одну систему начальника. Ці зв'язки начальника та підлеглого формують дерево зв'язків, що називається деревом фіксації транзакцій (*transaction's commit tree*). Залучені менеджери ресурсів також є членами commit tree. Вони, зазвичай, мають підлеглий зв'язок до їх локального менеджера транзакцій. Зв'язки підлеглих та начальників відносяться тільки до відповідної транзакції. Тобто менеджер транзакцій А може бути начальником для менеджера транзакцій В для транзакції Т1, при цьому будучи підлеглим до менеджера транзакцій В для транзакції Т2.

Коли розподілена транзакція підтверджується чи відміняється, до commit tree посилаються повідомлення про підготовку, підтвердження чи невдачу. Будь-який вузол дерева може в односторонньому порядку відмінити транзакцію у будь-який час. Після того, як вузол здійснив виконання своєї частини трансакції, він залишається у стані очікування остаточного підтвердження (in-doubt), поки координатор підтвердження (commit coordinator) не передасть йому команду підтвердити чи відмінити транзакцію. Центральний менеджер транзакцій дерева фіксації називається глобальним координатором фіксації (*global commit coordinator).* Цей координатор приймає рішення підтвердити чи відмінити транзакцію і ніколи не знаходиться у стані очікування остаточного підтвердження.

Якщо комп'ютер відказує і перезавантажується, менеджер транзакцій на цьому комп'ютері намагається визначити результат усіх транзакцій, які очікують підтвердження та у яких він приймав участь. Локальний менеджер транзакцій зчитує свій лог-файл, щоб визначити результати транзакцій, для яких він був координатором фіксації та виконує одну із наступних дій.

1. Якщо він підпорядкований менеджеру - начальнику (superior transaction manager), він визначає, чи він був попередньо оповіщений про результат транзакції менеджером начальником.
2. Якщо локальний менеджер транзакцій знаходиться у стані очікування остаточного підтвердження визначеної транзакції, він запитує свого начальника щодо отримання результату транзакції.

Менеджер транзакцій також відповідає на запити від менеджера - начальника щодо транзакцій в стані очікування. Це здійснюється подібно протоколу, якому менеджери транзакцій та менеджери ресурсів слідують при перезавантаженні. Менеджер транзакцій визначає результат кожної транзакції, що знаходиться в стані очікування остаточного підтвердження, та інформує менеджер ресурсів про результат транзакції.

Відкази вузлів системи чи мережі можуть залишити транзакції у непевному стані очікування остаточного підтвердження на тривалий час.

Поки транзакція знаходиться у непевному стані, ресурси, модифіковані транзакцією блокуються і являються недоступними для інших. В системі Windows системний адміністратор може використовувати адміністративний інструмент *Component Services* для вирішення щодо непевних транзакцій.

Відказостійкість кластеру. У разі функціонування системи обробки розподілених транзакцій у кластерному режимі, тільки один вузол у кластері може мати запущеною службу менеджера транзакцій DTC. Якщо вузол, що виступає в якості власника ресурсів відмовить, сервіс управління кластером автоматично призначить в якості власника ресурсу інший вузол у кластері, у разі відмови вузла, виконуючого роль менеджера транзакцій DTC – ця служба автоматично перезавантажиться на іншому вузлі у кластері.

Під час цієї операції ресурс DTC та залежні від нього ресурси автоматично переміщуються до іншого власника. Щойно перезавантажившись, менеджер транзакцій DTC зчитує лог-файл DTC зі спільного диску для того, щоб визначити результати транзакцій, які виконувалися під час відказу. Менеджери ресурсів знову підключаються до менеджеру транзакцій DTC та виконують відновлення для того, щоб визначити результат сумнівних транзакцій. Клієнтські додатки також перепідключаються до DTC.

Наприклад (рис.19), менеджер транзакцій DTC активний на Системі B. Програма додатку та менеджер ресурсів на Системі A викликають DTC-проксі. DTC-проксі на Системі A перенаправляє усі виклики DTC на менеджер транзакцій DTC на Системі B. Якщо система B відмовить, менеджер транзакцій DTC на системі A бере управління на себе. Він зчитує лог-файл на спільному диску, виконує відновлення і потім працює в якості менеджера транзакцій для кластеру.

Якщо використовується політика за умовчанням, вибір вузлу відказостійкості буде випадковим. Якщо необхідний більший контроль над тим, який вузол обирається для відказостійкості, ви можете використовувати інструмент Cluster Administrator для зміни політики.

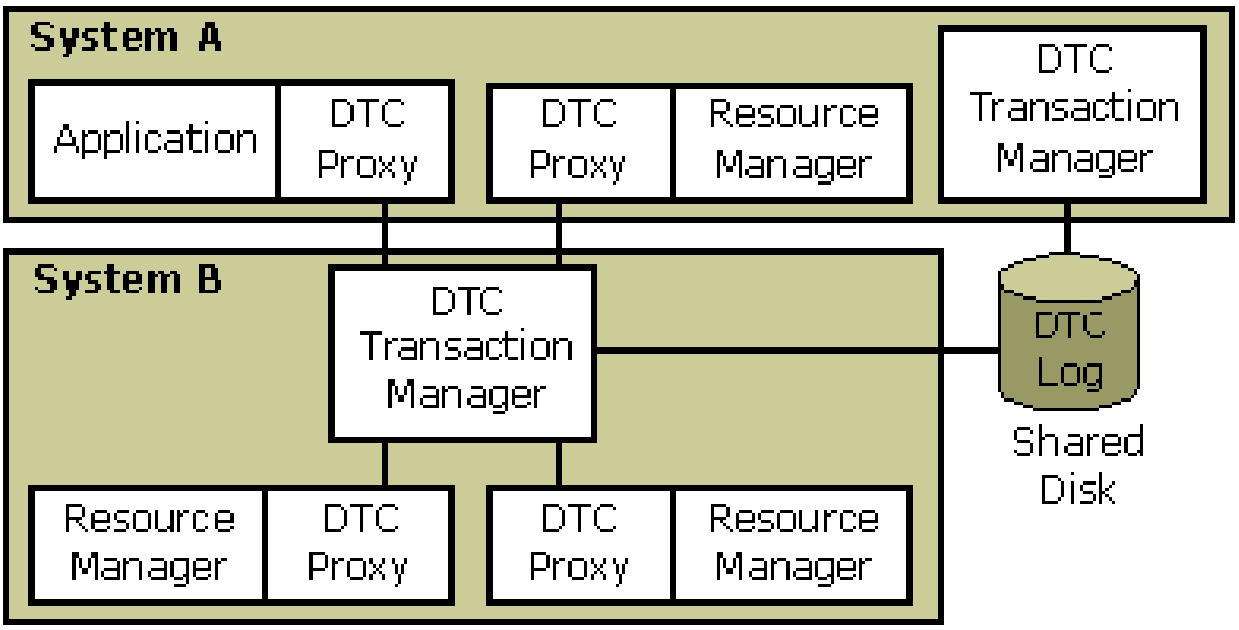



Рис. 19. Приклад відмовостійкого кластеру

У разі повернення до нормального стану власника DTC, що раніше відмовив, доцільно здійснити перенесення ресурсу DTC та його залежних ресурсів до відповідного вузла, на якому вони перебували до відказу. Такий процес перенесення ресурсу DTC та його залежностей має назву *відновлення (failback).* За умовчанням, ресурс DTC та його залежності залишаються на вузлі, до якого вони були переміщені під час процесу подолання відказу. Для автоматичного налаштування відновлення використовується інструмент Cluster Administrator.

### 4.3.2. Компонентно-орієнтована система обробки транзакцій

Система обробки транзакцій MTS **(*Microsoft Transaction Server)*** – призначена для побудови, впровадження та адміністрування потужних (robust) серверних Internet та intranet додатків. MTS складається з: середовища виконання (*run-time environment*); проводника (*MTS Explorer*) – продукту для управління компонентами додатків (application components); інтерфейсів програмування (*application programming interfaces*) та дозаторів ресурсів (*resource dispensers*) , які призначені для управління масштабованістю та стійкістю додатків. Дозатори ресурсів – це сервіси, які управляють оперативною інформацією загального доступу, наприклад, про зв'язок компонентів,

про стан менеджера ресурсів. Дозаторам ресурсів доступні динамічні об'єднання ресурсів (pools of resources) таких, як підключення до бази даних (database connections), підключення до мережі (network connections), підключення до черг (queues), потоків, блоків пам'яті. Завдяки взаємодії додатків з дозаторами ресурсів підвищується продуктивність роботи.

Модель програмування MTS надає середовища для побудови компонентів, відповідаючи за бізнес-логіку додатка. Середовище виконання (MTS run-time) – це платформа-посередник (middle-tier platform) для виконання цих компонентів . MTS Explorer дозволяє здійснювати регістрацію та управління роботою компонентів в середовищі.

Розглянемо основні елементи MTS: компоненти додатків; середовище виконання (MTS executive); серверні процеси, які обслуговують запити від клієнтів; менеджери ресурсів; дозатори ресурсів; координатор розподілених транзакцій; провідник компонентів (MTS explorer).

Компоненти додатків відповідають за бізнес логіку програмних систем – реалізовують бізнес правила, забезпечуючи відображення та трансформацію стану додатку. Наприклад, для online-банку записи в одній чи декількох БД відображають стійкий стан справ, такий як кількість грошей на рахунку. Компоненти додатків оновлюють стан для того, щоб відобразити такі зміни, як дебети (борги) та кредити.

MTS захищає розробників від складних проблем, пов'язаних з управлінням обробкою розподілених транзакцій, дозволяючи їм сфокусуватися на реалізації бізнес-функцій. Для компонентів, які працюють у середовище MTS, за виконання транзакцій відповідає середовище, таким чином, розробники можуть писати додатки не враховуючи при цьому специфіку ізольованості транзакцій та інші деталі. MTS керує паралелізмом, об'єднанням ресурсів, безпекою, управлінням контекстом та іншими складностями системного рівня. Система транзакцій, яка працює у співпраці із серверами БД та іншими типами менеджерів ресурсів, пересвідчується, що

паралельні транзакції атомарні, консистентні, мають належну ізольованість та у тому, що, будучи одного разу підтверджені, зміни є стійкими.

MTS також робить легшим процес будування розподілених додатків, забезпечуючи прозорість місць розташування (location transparency). Компонент MTS може бути завантажений у клієнтський процес додатку (in-process application component) або в окремий процес серверного середовища на клієнтському комп'ютері (local application component), чи на іншому комп'ютері (remote application component).

Компоненти MTS та COM. Компоненти MTS є внутрішньо-процесними серверними компонентами COM (in-process server components), що знаходяться у бібліотеках DLL. COM – специфікація, яка описує процес взаємодії компонентів через інтерфейси, при цьому доступ до інтерфейсів здійснюється з застосуванням методу QueryInterface, з визначенням при цьому часу життя для покажчика (pointer lifetime) через підрахунок числа посилань (reference counting), та повторним використанням об'єктів через агрегацію (COMDefinition). Вони відрізняються від інших COM-компонентів у тому, що виконуються у середовищі виконання MTS. Такі компоненти можна створювати використовуючи продукти Visual Basic, Visual C++, Visual J++, чи будь-якого ActiveX-сумісного інструменту розробки.

MTS накладає спеціальні вимоги на компоненти поверх тих, що вимагаються технологією COM. Це дозволяє MTS забезпечувати компонентам сервіси, які у іншому випадку були б неможливими. Сюди входять підвищена масштабованість та надійність, а також спрощене управління системою.

Диспетчер MTS (MTS Executive). Диспетчер MTS – це динамічно завантажувана бібліотека (DLL), що забезпечує сервіси виконання для компонентів MTS, включаючи потоки та управління контекстом. Ця DLL завантажується у процеси, на які відображуються компоненти додатку, і працює у фоновому режимі. Як вже було сказано вище, MTS також містить набір дозаторів ресурсів, що спрощує доступ до спільних ресурсів у серверному процесі.

Серверні процеси (Server Processes). Серверний процес – це системний процес, що утримує виконання компоненту додатку. Серверний процес може містити багато компонентів та може обслуговувати десятки, сотні, а потенційно і тисячі клієнтів. Можна налаштувати, щоб декілька серверних процесів виконувалися на одному комп'ютері. Кожен серверний процес представляє собою окремі границю довіри (trust boundary) та область локалізації несправностей (fault isolation domain).

Інші середовища процесів можуть також утримувати компоненти додатків. У результаті існує можливість розгортати додатки, що відповідають різноманітним вимогам розподілення, продуктивності та локалізації несправностей. Наприклад, можливо налаштувати, щоб компоненти MTS завантажувалися прямо на Microsoft Internet Information Server (IIS).

Менеджери ресурсів (Resource Managers). Менеджер ресурсів – це системний сервіс, що управляє так званими стійкими даними (durable data). Стійкість даних забезпечується як їх зберіганням на диску (наприклад, в базі даних), так і їх несуперечністю, узгодженістю відповідно до системних або бізнес правил. Серверні додатки використовують менеджери ресурсів для утримання стійких даних додатку, таких, наприклад, як запис інвентарю, поточні замовлення та рахунки по заборгованості. Менеджери ресурсів працюють у співробітництві із Microsoft Distributed Transaction Coordinator для можливості гарантування атомарності та ізольованості транзакцій. MTS підтримує менеджери ресурсів, такі як Microsoft SQL Server версії 6.5 та вище, що виконують протокол OLE Transactions.

Менеджер ресурсів реєструє себе із локальним менеджером транзакцій (local transaction manager) і чекає на виконання запитів від програми додатку. Коли надходить запит із новим об'єктом транзакції, менеджер ресурсів залучається (enlist) до транзакції, викликаючи метод Enlist. Завдяки цьому менеджер ресурсів засвідчує, що він отримує зворотні виклики від менеджера транзакцій, коли транзакція підтверджується чи відміняється. Тоді менеджер ресурсів виконує запити транзакції. Менеджер ресурсів зберігає необхідну

інформацію для того, щоб або відмінити, або заново виконати транзакцію, використовуючи лог-файл для фіксації змін або утримуючи декілька версій даних. Коли програма додатку підтверджує транзакцію, менеджер транзакцій ініціює протокол двох-етапної фіксації. Якщо менеджер ресурсів не може успішно здійснити обробку своєї частини транзакції, він повідомляє менеджер транзакцій, що він не готовий, і менеджер транзакцій відміняє розподілену транзакцію. Якщо менеджер ресурсів здатен приготуватися, він говорить менеджеру транзакцій, що він готовий і чекає рішення менеджеру транзакцій – підтвердження чи відміни транзакції.

*Дозатори ресурсів (Resource Dispensers)*. Дозатори ресурсів управляють нестійкими спільними даними на стороні компонентів додатку у рамках процесу. Дозатори ресурсів схожі на менеджерів ресурсів, але без гарантії стійкості. MTS забезпечує два типи керівників ресурсів: ODBC resource dispenser та Shared Property Manager.

ODBC resource dispenser управляє наборами підключень до БД ( database connections) для компонентів MTS, які використовують стандартні ODBC інтерфейси, швидко і ефективно виділяючи підключення для об’єктів. Підключення автоматично залучаються до транзакцій об’єкта, і дозатор ресурсу може автоматично відновити підключення та використовувати його заново. ODBC 3.0 Driver Manager – це дозатор ресурсів ODBC, компонент якого встановлюється разом із MTS.

Shared Property Manager забезпечує синхронізований доступ до визначених у додатку, процесних властивостей (змінних). Наприклад, це може застосовуватися для лічильника відвідувань веб-сторінки чи підтримки загального стану, доступного для різних процесів, багатокористувальницької гри.

### 4.3.3. Менеджер розподілених транзакцій

Microsoft Distributed Transaction Coordinator (MS DTC) – це системний сервіс, що координує обробку розподілених транзакцій. MS DTC виконує двох-етапний протокол фіксації для того, щоб пересвідчитися, що результат

транзакції (підтвердження чи відкат) є консистентним поміж усіх менеджерів ресурсів, залучених до транзакції. MS DTC забезпечує атомність незалежно від відказів. DTC підтримує менеджери ресурсів, що реалізують транзакції OLE, протоколи X/Open XA та LU 6.2 Sync Level 2.

MTS Explorer може використовуватися для розгортання компонентів додатків, для перегляду та зміни елементів у середовищі виконання MTS.

Менеджер транзакцій створює об'єкти транзакцій та управляє їх атомарністю та стійкістю. Додаток запитує створення об'єкту транзакції, викликаючи метод Begin Transaction менеджера. Коли менеджер ресурсів вперше приймає участь у транзакції, він викликає метод Enlist, щоб залучитися до транзакції. Менеджер транзакцій відслідковує усі менеджери ресурсів, які залучені до процесу обробки транзакції. Тут може виникнути один із наступних випадків: додаток або підтверджує транзакцію, або відміняє її; менеджер ресурсів відміняє транзакцію; виникає помилка.

Методи Commit та Abort також можуть бути викликані об'єктами транзакцій. Коли менеджеру транзакцій необхідно зафіксувати транзакцію, він ініціює протокол двох-етапної фіксації. Під час першого етапу він просить усі залучені менеджери ресурсів приготуватися. Під час другої фази менеджер транзакцій інформує менеджерів ресурсів про те, підтверджена чи відмінена дана транзакція.

Менеджер транзакцій утримує лог-файл у сховищі на диску, в якому він записує інформацію про запуски транзакцій, залучення та рішення про фіксацію. Під час нормальної роботи менеджер транзакцій тільки записує у лог, але у разі відмови він зчитує лог, щоб потім відновити свій останній стан.

DTC використовує оптимізацію протоколу двох-етапної фіксації для передбачення відкату, але DTC також підтримує наступні два варіанта оптимізації: Read-Only Commit Optimization та Delegated Commit Optimization.

*Read-Only Commit Optimization.* DTC дозволяє менеджерам ресурсів, які можуть тільки зчитувати, але не змінювати захищені транзакцією дані, повертати повідомлення “read-only” на першому етапі виконання розподіленої транзакції. У цьому випадку, менеджер транзакцій не доставляє оповіщення на другому етапі до менеджера ресурсів. Процедура read-only також використовується менеджерами транзакцій для того, щоб оптимізувати підтвердження частин дерева транзакцій (subtrees). Менеджер транзакцій повертає повідомлення "read-only", якщо всі підпорядковані до нього менеджери ресурсів і менеджери транзакцій повертають повідомлення "read-only". Це означає, що даному менеджеру транзакцій та частини дерева, яке він контролює, не потрібно отримувати оповіщення на другому етапі фіксації. Використання цієї оптимізації усуває необхідність певних записів у лог-файл, а також дозволяє швидке звільнення блокувань ресурсів.

*Delegated Commit Optimization.* DTC дозволяє делегувати обов’язки координатора фіксацій від координуючого менеджера транзакцій до одного із його підлеглих менеджерів транзакцій чи менеджерів ресурсів. Делегація прав координатора фіксацій виникає, коли координуючий менеджер транзакцій виявляє, що лише один підлеглий менеджер ресурсів чи менеджер транзакцій являється залученим до транзакції. Координуючий менеджер транзакцій делегує обов’язки координатора фіксацій до менеджера ресурсів, коли дійсні обидві наступні умови: жоден менеджер транзакцій не залучений до певної транзакції, тільки один менеджер ресурсів залучений до цієї транзакції.

У цьому випадку, координуючий менеджер транзакцій посилає повідомлення "delegated commit" на місце повідомлення першого етапу. Повідомлення "delegated commit" передає обов’язки координатора фіксацій від поточного координуючого менеджера транзакцій до його підлеглого менеджера транзакцій. Ця оптимізація застосовується рекурсивно – менеджер транзакцій, що отримує обов’язки координатора фіксацій, може делегувати ці обов’язки до свого підлеглого менеджера ресурсів чи

менеджера транзакцій. Використання цієї оптимізації усуває необхідність певних записів у лог-файл та дозволяє швидке звільнення блокувань на певних ресурсах.

Розглянемо процедури поширення та експорту транзакцій.

Транзакція, розпочата в одному процесі, може бути поширена на інший процес. Менеджери ресурсів, побудовані на клієнт-серверній архітектурі, мають серверну і клієнтську частини. Серверна частина залучається до транзакції так, що вона може приймати участь у двох-етапній фіксації. Проте, оскільки DTC - транзакція могла бути ініційована на клієнтській стороні, клієнт-серверні менеджери ресурсів використовують DTC-пристосований механізм для поширення транзакції від клієнтської сторони до серверної.

Для поширення транзакції проксі-менеджер ресурсів формує функцію, яку додаток може викликати для залучення менеджеру ресурсів до транзакції. Після отримання транзакції, до якої необхідно залучитися, проксі-менеджер ресурсів використовує інтерфейс ITransactionExport для поширення транзакції на координатор транзакцій.

Клієнтський додаток отримує інтерфейс ITransaction, викликаючи метод ITransactionDispenser::BeginTransaction. Клієнтський додаток викликає метод IResourceManager2::Enlist2, що передає йому транзакцію, до якої він намагається залучити певний менеджер ресурсів. У методі IReso-?rceManager2::Enlist2, проксі-менеджер ресурсів виконує наступне: викликає ITransactionExport::Export, передаючи йому інтерфейс ITransaction, наданий клієнтом; якщо попередній крок закінчується успішно, проксі- менеджер ресурсів отримує cookie транзакції, викликаючи ITran-?actionExport::GetTransactionCookie; проксі-менеджер ресурсів посилає cookie транзакції до менеджера ресурсів через комунікаційний канал, що використовується для передачі усіх повідомлень між проксі та сервером.

Після того, як менеджер ресурсів отримує cookie транзакції, він викликає ITransactionImport::Import, надаючи йому cookie, отриманий від свого проксі. Цей виклик переводить cookie у інтерфейс ITransaction, і, якщо він за-

кінчується успішно, менеджер ресурсів отримує інтерфейс для ініційованої клієнтом транзакції.

### 4.3.4. Інтерфейси доступу до баз даних із MTS

Основними ресурсами, які застосовуються в розподілених транзакціях, є бази даних. Розглянемо варіанти інтерфейсу доступу до БД для додатків MTS. З метою доступу до БД можливо використовувати або ODBC API (Open Database Connectivity Application Programming Interface) для доступу до менеджерів ресурсів, або модель доступу до даних, що функціонує над рівнем ODBC. Оскільки ODBC version 3.0 Driver Manager – це дозатор ресурсів MTS, дані, до яких звертаються через ODBC, автоматично захищаються транзакцією об'єкта. Для об'єктних транзакцій ODBC-сумісна БД має підтримувати наступне.

Драйвер ODBC бази даних повинен бути потокобезпечним (thread safe). Це означає, що додаток драйверу може маніпулювати загальними даними з гарантією безпечного виконання декількох потоків одночасно. Він також повинен мати змогу з метою підключення до драйверу одного потоку, використовувати підключення із іншого потоку, а також відключення з іншого потоку.

Якщо ODBC використовується із середини транзакційного компоненту, тоді драйвер ODBC повинен також підтримувати атрибут зв'язку SQL_ATTR_ENLIST_IN_DTC і контролюючи, який дозатор ODBC Driver Manager залучає підключення до транзакції .

В таблиці 8 надаються вимоги БД для повної підтримки MTS.

Якщо всередині драйверу виникає порушення доступу до пам'яті у процесі mtx.exe після 60 секунд бездіяльності, це вказує, що драйвер ODBC не є потокобезпечним чи вимагає схожість потоків. Помилка виникає у драйвері, коли неактивні з'єднання відключаються.

Вимоги БД для повної підтримки MTS

Таблиця 8

| Вимоги | Описання | Ресурси (якщо доступні) |
|---|---|---|
| Підтримка специфікації транзакцій OLE чи підтримка XA протоколу. | Дозволяє пряму взаємодію із DTC. Для взаємодії із DTC використовується XA Mapper. | MTS SDK у Microsoft Platform SDK. |
| Драйвер ODBC | Вимоги платформи до серверних компонентів MTS. | ODBC version 3.0 SDK |
| Підтримка ODBC драйверу для виклику ODBC version 3.0 SetConnectAttr SQL_ATTR_ENLIST_IN_DTC. | MTS використовує цей виклик для передачі ідентифікатору транзакції до драйверу ODBC. ODBC драйвер тоді передає ідентифі-катор до процесору бази даних. | ODBC version 3.0 SDK |
| Повністю потокобезпечний драйвер ODBC. | Драйвер ODBC повинен мати змогу керувати паралельними викликами від одного потоку у будь-який момент часу. | ODBC version 3.0 SDK |
| Драйвер ODBC не повинен вимагати схожість потоків. | Драйвер ODBC повинен мати змогу підключення до драйверу із одного потоку, використовувати підключення із іншого потоку та відключитися з іншого потоку. | ODBC version 3.0 SDK |

В таблиці 9 надані найпоширеніші моделі доступу до даних, які підтримуються MTS.

Моделі доступу до даних, які підтримуються MTS.

Таблиця 9

| Інтерфейс | Описання |
|---|---|
| ActiveX™ Data Objects (ADO), Remote Data Service (RDS) або Advanced data connector (ADC) | ADO пропонує одну поширену модель програмування для доступу до даних. ADO включає в себе можливість передавати результати запитів (Recordsets) поміж сервером та клієнтом та можливість передавати оновлені Recordsets від клієнта до сервера, використовуючи RDS. |
| OLE DB | OLE DB – це низькорівневий інтерфейс, що забезпечує однорідний доступ до будь-яких табличних джерел даних. Викликати інтерфейс OLE DB прямо із додатку, розробленого з застосуванням середовища Visual Basic, неможливо. Клієнтський додаток Visual Basic може отримати доступ до джерела даних OLE DB тільки використовуючи ADO. |
| Open DataBase Connectivity (ODBC) | ODBC – це визнаний стандартний інтерфейс для реляційних джерел даних. ODBC швидкий і забезпечує універсальний інтерфейс, який не оптимізований для будь-яких специфічних джерел даних. |
| Remote Data Objects (RDO) | RDO – це тонкий інтерфейс об'єктного рівня для ODBC API. Він спеціально розроблений для доступу до віддалених реляційних джерел даних ODBC. |

Діаграма на рис.20 показує як компоненти MTS взаємодіють із різними інтерфейсами доступу до даних. ADO спеціально розроблено для реляційних чи ISAM ( Indexed Sequential Access Method) баз даних в якості об'єктного інтерфейсу для будь-яких джерел даних. ADO може звертатися до реляційних баз даних, ISAM, текстових, ієрархічних та будь-яких інших типів ресурсів даних на протязі часу функціонування провайдера доступу до даних. ADO побудований навколо набору ключових функцій, який усі джерела даних повинні виконувати. ADO може звертатися до рідних джерел даних OLE DB, включаючи спеціальний провайдер OLE DB, що забезпечує доступ до драйверів ODBC. ADO поставляється із OLE DB Software Development Kit (SDK).

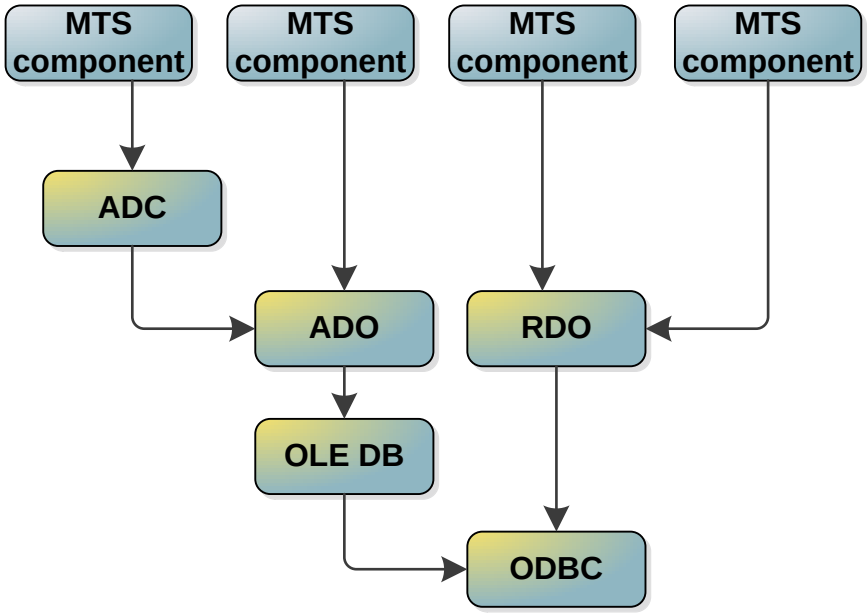



Рис.20. Взаємодія компонентів із різними інтерфейсами доступу до даних

### 4.3.5. Засоби .NET для обробки розподілених транзакцій

У Windows системний сервіс DTC керує транзакціями поміж компонентами, процесами та машинами, використовуючи протокол *OLE Transactions (OleTx)*. Стосовно .NET найбільш розповсюджений та простий шлях використання DTC транзакцій – це використання бібліотеки Enterprise Services. Приклад використання Enterprise Services транзакцій наведено нижче.

```
Using System.EnterpriseServices;
[Transaction]
public class MyComponent : ServicedComponent
{
    [AutoComplete]
    public void MyMethod()
    {
        /*Interact with other serviced components
        and resource managers */
    }
}
```

.NET Enterprise Services пропонують декларативну модель програмування: будь-який клас, що походить від абстрактного класу ServicedComponent може використовувати атрибут Transaction. Цей атрибут засвідчує, що коли викликається будь-який метод класу, цей метод виконується у середині транзакційного середовища. Середовище (context) – це внутрішня область дії компоненту. .NET перехоплює виклики, що надходять до цієї області, і починає транзакцію від імені об'єкта. Не потрібно жодних явних залучень ресурсів. Такі ресурси називаються транзакційними менеджерами ресурсів і багато із популярних комерційних баз даних та довготривалих ресурсів (таких як Microsoft Message Queue) є менеджерами ресурсів. Все що повинен зробити об'єкт – це повідомити .NET щодо підтвердження чи відміни транзакції. Коли це можна зробити використовуючи методи допоміжного класу ContextUtil, ви також можете виконати це декларативно, через атрибут AutoComplete. Якщо не було повернено жодних виключень, атрибут AutoComplete підтвердить транзакцію, а якщо виникло виключення, транзакція буде відмінена.

Не дивлячись на те, що декларативна модель програмування пропонує значні вигоди у продуктивності, вона має свої недоліки:

- Примусове наслідування від ServicedComponent займає цінний простір базового класу, який зазвичай резервується для моделювання внутрішніх додатків.
- Використання Enterprise Services транзакції завжди означає використання розподіленої DTC транзакції, навіть коли залучені один об'єкт і один ресурс. Протокол двох-етапної фіксації має свої витрати і на рівні менеджера транзакцій і на рівні ресурсів, оскільки ресурс повинен продовжувати логувати свої операції. Накладні витрати можуть викликати погіршення виконання у порівнянні із явним управлінням транзакціями.
- Зв'язок із використанням Enterprise Services у моделі COM+, що відлякує багатьох розробників .

• Enterprise Services транзакції тісно поєднані із Enterprise Services стратегіями управління екземплярами. Усі транзакційні об'єкти також активуються just-in-time, і виникають деякі проблеми, коли необхідно комбінувати транзакції із організацією об'єктних пулів. Тоді як таке поєднання добре підходить до масштабованих додатків, для всіх інших воно змушує використовувати state-aware модель програмування, з чим розробники мають складності.

• Enterprise Services транзакції завжди потокобезпечні – у жодному випадку декілька потоків не можуть приймати участь в одній транзакції. Тоді як це значно полегшує управління транзакціями особливо у багатопоточному середовищі, у деяких крайніх випадках є певним обмеженням.

Фактично платформи .NET 1.0 та 1.1 прирівнюють механізм використання нерозподілених транзакцій до механізму з явним управлінням транзакціями, а використання розподілених транзакцій до декларативних транзакцій Enterprise Services. Неможна використовувати декларативну транзакцію без використання DTC транзакції. Також немає легкого способу, що забезпечує керований код для явного управління транзакціями, що використовує DTC. Вибір моделі програмування (явної чи декларативної) завжди означає і вибір менеджера транзакцій та навпаки.

Для вирішення щойно описаних проблем як з явною та і з декларативною моделями програмування .NET Framework Version 2.0 представляє два нові типи менеджера транзакцій – *Lightweight Transaction Manager* (LTM) та *OleTx Transaction Manager*.

*LTM* використовується для управління транзакцією усередині домену додатку, що залучає максимум один довготривалий ресурс. *OleTx Transaction Manager* управляє транзакціями поміж границями домену додатку (також поміж границями процесів та машин) чи будь-якими транзакціями, що залучає більше ніж один ресурс, навіть у тому ж домені додатку. OleTx transaction manager покладається на RPC для між-машинних викликів. LTM використовує тільки виклики всередині домену додатку, а відтак відзначається більшою продуктивністю ніж OleTx transaction manager. Застосовуючи механізм

System.Transactions розробникам не потрібно взаємодіяти з цими менеджерами транзакцій безпосередньо. Замість цього в області імен System.Transactions доступна загальна інфраструктура, що визначає: інтерфейси; фабрики транзакцій (transaction factories); класи поведінки (behavior classes); допоміжні класи (helper classes).

Подібно до Enterprise Services, System.Transactions менеджер ресурсу може автоматично включитися у транзакцію, яка управляється LTM чи OleTx. Основна перевага програмування із загальною областю керування транзакціями (System.Transactions) над спеціальним менеджером транзакцій складається в автоматичному виборі режиму виконання транзакції (promotion). Наприклад, якщо один об'єкт взаємодіє з одним тривалим ресурсом – це вимагає LTM, але якщо необхідно забезпечити транзакцію для іншого об'єкту у іншому домені додатку або для іншого менеджера ресурсів, транзакція автоматично перейде до управління менеджером транзакцій OleTx.

Простір імен System.Transactions містить класи, що дозволяють створювати власні додатки, в яких використовуються транзакції й диспетчери ресурсів. Зокрема, можна створювати транзакції й брати участь у транзакціях (локальних і розподілених) з одним або декількома учасниками.

Інфраструктура System.Transactions робить програмування транзакцій простим і ефективним у всій платформі, підтримуючи транзакції, що ініціюються в SQL Server, ADO.NET, MSMQ (Microsoft Message Queue) і координаторі розподілених транзакцій (DTC).

В основі System.Transactions лежить клас TransactionScope. При створенні екземпляра цього класу він створює поточну транзакцію (вона також називається транзакцією навколишнього середовища), до якої може прикріпитися будь-який диспетчер ресурсів. Наприклад, якщо створено екземпляр класу TransactionScope і при цьому відкрито з'єднання з диспетчером ресурсів, що за замовчуванням прикріплюється до транзакцій, це з'єднання буде включено в клас TransactionScope. Для його використання необхідно вказати

посилання на System.Transactions.dll. Далі об’єднуємо команди роботи з базою даних в оболонку, як показано в прикладі (рис. 21), і поміщаємо наприкінці транзакції метод Complete.

```
using(TransactionScope scope = new TransactionScope())
{
        /* Perform transactional work here */
        //No errors – commit transaction
        scope.Complete();
}
```

Якщо при виконанні будь-якої команди SqlCommand, поміщеної усередину оболонки, виникає виключення, потік команд пропускає блок операторів using класу TransactionScope, після чого клас TransactionScope видаляє сам себе й транзакція відміняється. Оскільки в коді використовуються оператори using, об'єкти SqlConnection і TransactionScope видаляються автоматично. У такий спосіб одержуємо модель транзакцій, здатну обробляти виключення, очищати свої службові елементи й контролювати виконання й скасування команд.

Існує можливість зміни настроювань за замовчуванням для класу TransactionScope. Можна встановити рівень ізоляції й час очікування виконання транзакції в об'єктах TransactionScope, створивши об'єкт Trans-?ctionOptions. Клас TransactionOptions має властивість IsolationLevel, що дозволяє змінювати встановлений за замовчуванням рівень ізоляції (Seria-?izable). Наприклад, можна встановити рівень ReadCommitted або рівень Snapshot, що з'явився в SQL Server 2005. Більшість механізмів баз даних намагаються використовувати пропонований рівень ізоляції, але можуть вибрати й інший рівень. Для зміни встановленого за замовчуванням часу очікування в одну хвилину в класі TransactionOptions використовується властивість TimeOut.

При використанні простору імен System.Transactions дуже важливо розуміти, як і коли завершується транзакція. Якщо об'єкт TransactionScope не

видаляється належним чином, транзакція залишається відкритою доти, поки цей об'єкт не видаляється збирачем сміття або поки не закінчується час очікування. Залишати транзакції відкритими не рекомендується ще й тому, що відкриті транзакції блокують ресурси диспетчера ресурсів. Для демонстрації розглянемо наступний зразок коду:

```
TransactionScope ts = new TransactionScope();
SqlConnection cn2005 = new SqlConnection(cnString);
SqlCommand cmd = new SqlCommand(updateSql1, cn2005);
cn2005.Open();
cmd.ExecuteNonQuery();
cn2005.Close();
ts.Complete().
```

Цей код створює екземпляр об'єкту TransactionScope і приєднується до транзакції при відкритті з'єднання SqlConnection. Якщо все проходить нормально, команда виконується, з'єднання закривається, транзакція завершується й видаляється. Якщо виникає виключення, керування потоком пропускає етапи закриття з'єднання SqlConnection і видалення TransactionScope, у результаті чого транзакція залишається відкритою довше, ніж це було потрібно. Основне завдання полягає в тому, щоб правильно видалити клас Trans-?ctionScope так, щоб транзакція якнайшвидше була завершена або скасована. Цю проблему можна вирішити двома простими способами: використати блок try/catch/finally або оператор using. Можливо декларувати об'єкти за межами блоку try/catch/finally, додати блок у код try для створення об'єктів і виконання команди й помістити видалення класу TransactionScope і закриття з'єднання SqlConnection у блок finally. Завдяки цьому транзакція буде вчасно закриватися.

Краще застосовувати оператор using, оскільки він непрямим образом також створює блок try/catch. Оператор using гарантує видалення класу TransactionScope навіть у випадку, якщо посередині виконання блоку коду відбудеться виключення. Він забезпечує виклик процедури Dispose() для кла-

су TransactionScope при виході з блоку. Це важливо, оскільки перед видаленням класу TransactionScope транзакція завершується. Після завершення транзакції клас TransactionScope перевіряє, чи викликався метод Complete(). Якщо цей метод викликався, транзакція фіксується, а якщо ні – виконується скасування. Наведений вище код можна переписати в наступний спосіб:

```
using (TransactionScope ts = new TransactionScope()) {
using (SqlConnection cn2005 = new SqlConnection(cnString)
{
SqlCommand cmd = new SqlCommand(updateSql1, cn2005);
cn2005.Open();
cmd.ExecuteNonQuery();
}
ts.Complete();}
```

Слід підкреслити, що застосовано оператор using для класу Trans-?ctionScope і для об'єкта SqlConnection. Завдяки цьому обидва об'єкти будуть правильно й швидко вилучені при виявленні виключення. При відсутності виключення об'єкти видаляються по закінченні блоку коду з оператором using (на останній фігурній дужці).

Велика перевага простору імен System.Transactions полягає в підтримці полегшених транзакцій. Полегшені транзакції не використовують координатор розподілених транзакцій MS DTC доти, поки це не потрібно. Якщо транзакція є локальною, вона залишається полегшеною. Якщо транзакція стає розподіленою й використовує інший диспетчер ресурсів, полегшена транзакцію перетворюється в повністю розподілену транзакцію і використовує DTC.

Використання DTC, коли це необхідно в розподілених сценаріях, потребує велику кількість ресурсів. Тому краще уникати його використання, якщо це можливо. Починаючи з версії SQL Server 2005 реалізується підтримка полегшених транзакцій, на відміну від попередніх версій SQL Server. Якщо планується використовувати розподілені транзакції, System.Transactions буде розумним вибором. Крім того, якщо диспетчер ресурсів підтримує полегшені

транзакції, System.Transactions також буде відмінним варіантом. Однак не завжди кращий вибір полягає в тім, щоб використовувати простір імен System.Transactions для об'єднання всіх команд бази даних.

Наприклад припустимо, що додаток відправляє кілька команд базі даних, яка не підтримує полегшені транзакції. Згідно правилам ці операції необхідно об'єднати в одну транзакцію, щоб забезпечити їхню цілісність. Якщо команди в складі транзакції призначаються для однієї бази даних, транзакції ADO.NET будуть ефективніше, ніж транзакції System.Transactions, оскільки транзакції ADO.NET у цьому випадку не задіють DTC. Якщо ж у додатку дійсно використовуються розподілені транзакції, System.Transactions буде добрим варіантом вибору.

System.Transactions працює фактично з усіма диспетчерами ресурсів, але реальні переваги одержують тільки ті диспетчери ресурсів, які підтримують полегшені транзакції. Велика перевага полегшених транзакцій полягає в тім, що вони можуть визначати, чи треба їм перетворюватися в розподілені транзакції. Якщо необхідно змінити параметри нумерації Trans-?ctionScopeOptions, можна використовувати сім існуючих перевантажених конструкторів TransactionScope (таблиця 10). Засоби нумерації дозволяють контролювати взаємодію вкладених транзакцій.

Параметри нумерації TransactionScopeOptions

Таблиця 10

| TransactionScopeOptions | Опис |
|---|---|
| Required | Якщо транзакція вже існує, ця транзакція буде до неї приєднана. У протилежному випадку буде створено нову транзакцію. |
| RequiresNew | Буде створено нову транзакцію. |
| Suppress | Якщо активна транзакція уже існує, ця транзакція не буде до неї приєднуватися і не буде створюватися нова транзакція. Цю опцію варто використовувати, коли потрібно залишити код за межами транзакції. |

Значенням за замовчуванням і найбільше часто використовуваним є значення TransactionScopeOption.Required. Якщо оболонка сконфігурована цим значенням і існує зовнішня транзакція, нова транзакція приєднається до зовнішньої. У протилежному випадку (зовнішньої транзакції не існує), створюється нова транзакція, що стає кореневою оболонкою. Коли використовується опція TransactionScopeOption.Required, код усередині TransactionScope не повинен залежати від того, чи створюється нова транзакція, або просто приєднується до існуючої. Він повинен працювати однаково для обох випадків, тому що немає можливості простежити який саме випадок виконується.

Якщо встановлено опцію TransactionScopeOption.RequiresNew, то завжди буде створюватися нова транзакція, оболонка якої буде кореневою. Ця опція корисна, коли потрібно виконати транзакцію зовні оболонки зовнішньої транзакції: наприклад, потрібно виконати loging або аудит операцій, або передати подію підписаному додаткові незалежно від того, коли буде виконана або скасована зовнішня транзакція.

Якщо область сконфігурована як TransactionScopeOption.Suppress, то вона ніколи не буде частиною транзакції, незалежно від того, чи існує зовнішня транзакція. Значення зовнішньої транзакції для такої області завжди буде встановлене в null. Ця опція корисна, коли операції, представлені секцією коду повинні бути виконані, і їх не варто скасовувати якщо відбудеться відкат транзакції. TransactionScopeOption.Suppress дозволяє включати секцію коду, що не відноситься до транзакції, усередину оболонки транзакції. Однак варто обережно застосовувати перетинання звичайних областей з областями транзакцій, тому що це може викликати порушення ізоляції й послідовності виконання. Не транзакціонні області можуть викликати помилки, які, однак, ніяк не відіб'ються на виконанні транзакції. Також варто дуже обережно використовувати опцію TransactionScopeOption. RequiresNew і перевіряти, щоб дві транзакції (зовнішня і створювана) не порушували черговість виконання транзакцій, якщо одна буде виконана, а друга скасована.

У наступному прикладі представлений об'єкт класу TransactionScope, що створює три нових об'єкти цього класу. Кожний об'єкт сконфигуровано різним значенням параметра TransactionScopeOptions. На рис. 22 графічно зображені отримані транзакції.

```
using(TransactionScope scope1 = new TransactionScope())
//Default is Required
{
    using(TransactionScope scope2 = new Trans-
?ionScope(TransactionScopeOption.Required))
    {...}
    using(TransactionScope scope3 = new Trana-
?tionScope(TransactionScopeOption.RequiresNew))
    {...}
    using(TransactionScope scope4 = new Trana-
?tionScope(TransactionScopeOption.Suppress))
    {...}
    ...
}
```

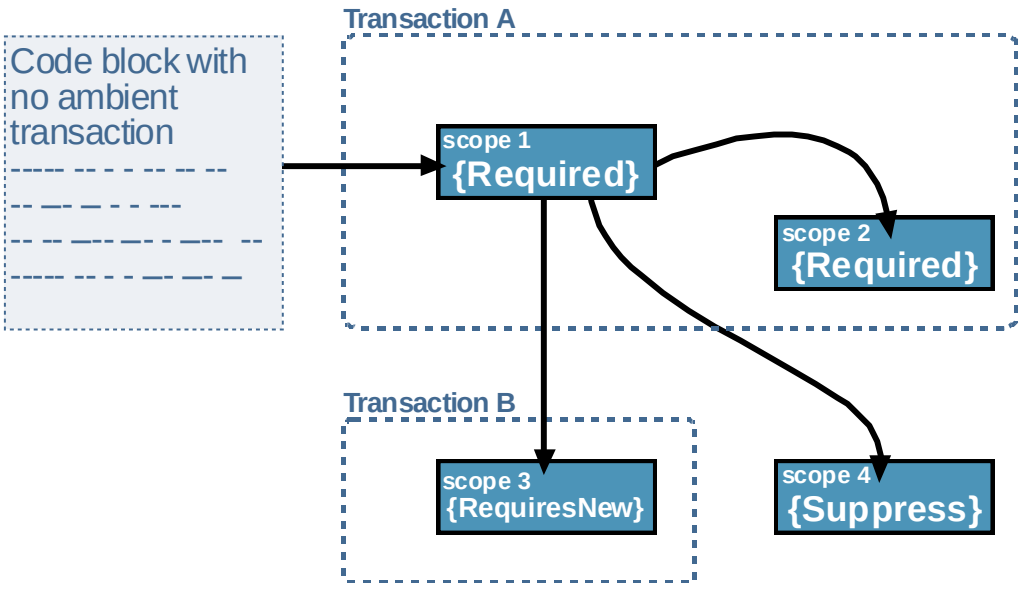



Рис. 22. Приклад використання різних опцій TransactionScope

На рис. 22 показано блок коду без зовнішніх транзакцій, що створює нову оболонку TransactionScope – scope1 з опцією TransactionScopeOption. Required. При цьому scope1 буде кореневою оболонкою: вона створює нову транзакцію (Transaction A) як зовнішню транзакцію. Далі scope1 створює три нових об'єкти з різними значеннями TransactionScopeOption. Scope2 сконфигурована як Required, і так як існує зовнішня транзакція, вона приєднається до Transaction A. Scope3 буде кореневою оболонкою для нової транзакції – Transaction B, а Scope4 не містить транзакцій.

Коли транзакції приєднуються до класу TransactionScope (при використанні параметра Required), для фіксування потрібне завершення всіх транзакцій. Якщо хоча б одна із транзакцій в одному класі TransactionScope не завершується за допомогою виклику ts.Complete, при видаленні зовнішнього класу TransactionScope відбувається її скасування.

Оскільки транзакції блокують цінні ресурси, дуже важливо швидко входити в транзакції й виходити з них. Найкраще створювати транзакції безпосередньо перед тим, як вони потрібні, відразу ж відкривати підключення щодо виконання команд, виконувати запити, завершувати транзакцію і видаляти її якомога швидше. Під час виконання транзакції варто уникати виконання непотрібного коду, що не відноситься до бази даних, щоб запобігти блокуванню ресурсів протягом більш тривалого часу, чим це потрібно.

## 5. Нетранзакційні розподілені бази даних і принцип BASE

Теоретичною базою для будь якої обчислювальної системи, яка спрямована на виконання програм та побудована за принципом Джона фон Неймана є математична модель «машини Тьюрінга», яка складається з наступних компонентів: нескінченої ленти (tape) з нескінченою кількістю комірок (cells), кожна з яких може містити один символ; голівки (header), що може зчитувати або записувати символ у комірку та може переміщуватися вліво/вправо; таблиці переходів між станами (state transition table), яка управляє голівкою. Таблиця переходів може бути задана з застосуванням різних мета-

фор описання даних та програм та на різних рівнях від процесора до Web-сервісу. Але це нічого не змінює концептуально – все, що в реальності робить система – це операції читання даних із сховища, перетворення відповідно до таблиці станів та запису проміжного або остаточного результату до сховища (reading-calculating-writing). Операції запису та читання являються базовими для систем любого роду від мікропроцесорів до хмарних сервісів (cloud services). Безумовно, операції зчитування/запису реальних сучасних систем відрізняються від модульованих машиною Тьюрінга, але можна говорити про деяку модель сховища даних (data model), яка являється метою для операцій читання та запису (the target of read/write operations ) та модель управління такими операціями або модель запитів на виконання таких операцій (query model). Модель запитів може розглядатися як інтерфейс програмування додатків (API) сконцентрований на операціях читання та запису при роботі з моделлю сховища. Модель сховища та модель запитів залежать одна від одної.

В сучасних системах критичною проблемою являється проблема масштабованості - потужності одного комп'ютера, навіть сервера, не вистачає для обробки запитів з відповідним рівнем якості. Спроби розширення системи без втрати централізованої моделі управління шляхом scale-up (vertical scaling, scale-in), тобто покупки нового більш потужного серверу або додавання процесорів, модулей пам'яті, дисків – призводять до того, що ефективність такої системи при відповідному зростанні обчислювального навантаження (кількості запитів) буде зменшуватися. Як правило у разі неефективності централізованої системи використовується підхід розподілення навантаження на множину незалежних комп'ютерів, реальних або віртуальних (scale-out, horizontal scaling). Базовим принципом при цьому являється «більше розподілення на потрібному рівні сервісу надає більшу його пропускну здатність». Але застосування моделі масштабування scale-out призводить до проблем у використанні стандартних реляційних транзакційних СУБД (РСУБД).

### 5.1. Підхід Scale-out для реляційних СУБД

Розглянемо ситуацію, в якої таблиці РСУБД розподілені між декількома комп'ютерами та кожна частина даних реплікується перед зберіганням для більшої доступності. Виконанання розділеної транзакції, коли задовольняються умови ACID, є важким для підходу scale-out.

Для задовільнення умови *атомарності*, протокол виконання транзакцій, такий як 2PC, повинен використовуватися у всіх системах, які відносяться до конктретної транзакції.

Для задовільнення умови *ізоляції* дані повинні бути заблоковані в загальному вигляді. Одиницею блокування може бути запис, таблиця або індекс.

Для задовільнення атрибутів *атомарності* та *ізольованості* в робочому середовищі, всі замки повинні відноситись до відповідної системи в момент, коли протокол транзакцій виконується; чим більше навантаження на систему, тим більше стає конкуренція замків. Інша проблема - ліміт реплікаційних та робочих даних при використанні scale-out.

У разі застосування реплікації master-slave (найбільш часто використовуємий метод реплікацій) швидкість процесу зворотньо-пропорційна кількості систем, які приймають участь в цьому методі реплікацій.

У разі застосування реплікації multi-master важко вирішити проблему зіткнення між процесами запису даних, коли є декілька майстрів (masters).

Проблеми РСУБД стають критичними у разі їх використання для обробки об'ємних Web-scale даних. РСУБД базується на використання структурованих даних. При цьому повинні виконуватися умови: властивості даних можуть бути визначені, а їх зв'язки встановлені заздалегідь; індекси можуть бути використані на наборах даних для прискорення запитів. РСУБД може, звичайно, мати справу з деякими порушеннями основних принципів. В зв'язку з використанням об'ємних наборів даних, стандартні механізми зберігання даних та доступу вимагають розширення: створюються денормалізовані таблиці, використовуються спадаючі обмеження. Але те, що залишаєть-

ся після цих модифікацій РСУБД, стає схожим на анти-РСУБД, що вже не базується на принципах транзакційних реляційних баз даних.

### 5.2. Теорема CAP та принцип BASE

Концепція нетранзакційних баз даних пов'язана з ім'ям професора Каліфорнійського університету в Берклі Еріка Брюєра, який у липні 2000 року [Brewer, 2000] запропонував теорему CAP, що використовується як теоретичне обґрунтування неминучості відмови від принципів ізольованості транзакцій в розподілених нетранзакційних системах управління базами даних в обмін на підвищення ступеня їх доступності і масштабованості. Лекція самого професора доступна за адресою http://www.cs.berkeley.edu/ ~ brewe?-/cs262b-2004/PODC-keynote.pdf

Теорема CAP (теорема Брюєра) – це евристичне твердження про те, що в будь-якій реалізації розподілених обчислень можливо забезпечити не більше двох з трьох наступних властивостей:

- узгодженість даних (consistency) - у всіх обчислювальних вузлах в один момент часу дані несуперечать один одному;
- доступність (availability) - будь-який запит до розподіленої системі завершується коректним відгуком;
- стійкість до розподілу (partition tolerance) - розщеплення розподіленої системи на декілька ізольованих секцій не призводить до некоректності відгуку від кожної із секцій.

Основною проблемою існуючих транзакційних СУБД є забезпечення стійкого стану збереженої інформації, згідно з принципом ACID. На відміну пропонується принцип BASE. Визначемо основні характеристики щодо цього принципу.

***Basically Available*** - базова доступність. Мається на увазі такий підхід до проектування додатків, щоб збій в деяких вузлах приводив до відмови в обслуговуванні тільки для незначної частини сесій при збереженні доступності в більшості випадків.

***Soft state*** - нестійкий стан. Передбачає можливість жертвувати довгостроковим зберіганням стану сесій (таких як проміжні результати вибірок, інформація про навігацію, контексти), при цьому концентруючись на фіксації оновлень тільки критичних операцій.

***Eventually consistent*** - узгодженість в кінцевому рахунку, означає можливість суперечності даних в деяких випадках, але при забезпеченні узгодження в практично найближчому майбутньому.

Необхідно враховувати, що РСУБД є більш зручною коли операції читання/запису потребують виконання принципів ACID. У більшості випадків, впровадження РСУБД та додаткової системи зберігання вирішить проблему розподілу робочого навантаження. Типовими рішеннями являються наступні.

- Розподіл зчитування через реплікації.
- Технологія scale-out може бути застосована в домені *OLTP* з використанням *MySQL Cluster* (https://www.mysql.com/ products/cluster/).
- Якщо продуктивність РСУБД не є дуже хорошою, то можна використовувати систему кешування, таку як Arcus (http://www.memcached.org/), для залежних від часу та часто використовуваних запитів.
- Якщо потрібно зберігати дані великих об'ємів, можна застосовувати окрему систему зберігання.
- Існують методи шардінгу (розподілу даних по вузлам кластеру), реалізовані розробниками.

**NoSQL** (часто інтерпретується як not only sql – "не тільки sql") – система спрямована на вирішення складнощів, які пов'язані з реалізацією масштабованих систем, та має набор властивостей: наявність вільної схеми, орієнтованої на обробку великих масивів даних, наявність простого API, задовільняє принципам BASE.

Такі системи ідеально підходять, якщо модель даних та вимоги до консистентності даних дозволяють використовувати NoSQL. Гнучкість має свою ціну. NoSQL усуває проблеми, які ісунють в РСУБД, та робить легкою роботу з об'ємними даними, але, в свою чергу, програє в цілісності транзакцій та

гнучкості индексування й запитів. Серед особливостей слід відзначити наступне.

### 5.2.1.Орієнтованість на роботу з документами

На відміну від реляційної моделі, згідно якої база даних є сукупністю таблиць (відношень атрибутів та кортежів або стовбців та рядків), пов'язаних з використанням ключових атрибутів,- при цьому структура таблиць диктує наявність та правила завдання значень усім атрибутам кортежу, модель документо-орієнтованої бази – це множина колекцій документів, при цьому кожен документ у колекції може мати свою власну структуру. Це робить базу більш гнучкою. Проте, основна частина даних повинна бути добре структурована, щоб уникнути повного хаосу при роботі з ними, чи при зміні структури бази даних. Головне в безструктурності архітектури є те, що вона не потребує встановлення та має мінімальні розходження з концепцією об'єктно-орієнтованого програмування. Дані зберігаються у вигляді пар key-value - одне значення-документ відповідає одному ключу. Як правило значення зберігається у формі BLOB. Існує також модель, яка забезпечує ADT (Absract Data Types) такі як список або набір, має можливість зберегти декілька значень для одного ключа. Отримання/внесення змін є дуже швидким (ідеально, коли розмір даних поміщається в сторінку пам'яті). Операції з декількома ключами можуть бути значно повільнішими, так як мережі передачі даних зазвичай працюють с затримками. Це оптимальна структура для зберігання документів таких, як JSON або XML.

### 5.2.2. Шардінг (sharding)

Так називається процес розподілу та зберігання різних «порцій» даних на різних машинах. Це робиться з метою зберігання та, при необхідності, обробки великих масивів даних без участі потужних серверних машин. Одиниця цих поділених даних називається *шард* (*shard* – англ., осколок). СУБД не

управляє шардами - це обов'язок розробника сервісу. Основними складнощами тут є наступні.

• Відображення шардів. В зв'язку з тим, що одна колекція може мати декілька шардів, потрібно знати, який шард відноситься до тієї чи іншої колекції. Місце локації всіх шардів повинні бути зазначені в розробленому додатку.

• Розподіл/перерозподіл шардів. Інформація, яка модифікується процесом розподілу/перерозподілу, має бути застосована в додатку. Оскільки шарди відрізняються по пропускній спроможності та розміру даних, то при додаванні або стиранні частини даних розробник повинен буде перерозподілити шарди.

MapReduce. Горизонтальна (scale-out) масштабованність зводиться до застосування кластеру однотипних або разнотипних вузлів, де кількість вузлів кластеру збільшується, коли збільшується навантаження. Деякі інфраструктури мають тисячі та навіть сотні тисяч серверів. Одним з ефектиних методів обробки даних розподілених на вузлах кластеру є модель MapReduce (відображення – збір результатів), який можна звести до двох фаз: на крокі Map відбувається попередня обробка даних; на крокі Reduce відбувається згортка попередньо оброблених даних. Таким чином, головний вузол отримує відповіді від робочіх вузлів та на їх основі формує остаточне рішення задачі.

Існує багато реалізацій систем NoSQL (150 – за даними сайту http://nosql-database.org/). Типовим представником таких систем є MongoDB. Розглянемо порівняння основних концепцій MongoDB з відповідними SQL.

• База даних MongoDB концептуально відповідає поняттю реляційної база даних. Сервер MongoDB зв'язується з відповідною IP-адресою та портом, і може містити визначену кількість баз даних, кожна з яких являється контейнером для інших сутностей.

• База даних може мати ряд «колекцій». Колекція може сприятися як «таблиця» з вільною структурою (schema-free).

• Колекції можуть ряд «документів». Аналогом документу в реляційній моделі є «рядок».

• Документ складається визначеного числа «атрибутів», які подібні до «стовпців» звичайної таблиці. Основною відмінністю є можливість зв'язування з атрибутами комплексних об'єктів (подібно об'єктно-орієнтованим базам даних).

• «Індекси» в MongoDB практично ідентичні індексам в реляційних БД.

• «Курсори» відрізняються від попередніх концепцій - коли ми робимо запит на отримання даних з MongoDB, у відповідь ми отримуємо курсор, за допомогою якого можливо виконувати операції над даними.

Отже, MongoDB складається з «баз даних», які складаються з «колекцій». «Колекції» складаються з «документів». Кожен «документ» складається з «полів». «Колекції» можуть бути проіндексовані, що значно покращує продуктивність роботи з даними колекцій, наприклад, сортування, вибірки. Отримання даних в MongoDB зводиться до отримання «курсору», який надає доступ до цих даних.

Основна відмінність стосовно елементів колекцій складається в наступному: згідно з документно-орієнтованою концепцією елементом колекції є документ - об'єкт, який має усі переваги об'єктно-орієнтованого уявлення (комплексні атрибути, що можуть використовуватися для задавання складних ієрархічних схем об'єктів).

Наприклад, ненормалізована схема документу Order може мати вигляд:

```
order
{
   "_id" : orderId,
   "user" : userInfo,
   "items" : [
        {
        "_id" : productId1,
        "name" : name1,
        "price" : price1
        },
        {
        "_id" : productId2,
        "name" : name2,
        "price" : price2
```

```
        },
        {
        "_id" : productId3,
        "name" : name3,
        "price" : price3
        }
        ]
}
```

В документі в колекції Items продукти задані у вигляді:

```
product
{
        "_id" : productId,
        "name" : name,
        "price" : price,
        "desc" : description
}
```

В реляційній моделі ми матимемо дві (у разі виключення довідника продуктів) таблиці і зв'язок типу «один заказ до багатьох продуктів».

Важливою особливістю є можливість зберігання в одній колекції документів з різною структурою, подібно до структур даних об’єктно-орієнтованого підходу (HashTable, Dictionary), до якого звикли розробники. Таким чином, документ містить набагато більше інформації ніж рядок і не вимагає нормалізації (розподілення частин документу по різним таблицям, зв’язаним ключами). Це впливає на швидкість виконання CRUD-операцій, пов’язаних з документом.

Шардінг в MongoDB забезпечується наступним чином: додаток звертається до mongos-процесу. Він, в свою чергу, до config-процесу та з'ясовує, в яких комірках знаходяться дані та в яких шардах знаходяться ці комірки. Дізнавшись номеру/номерів шарда, він звертається до них (mongod - процес) та отримує дані, повертая їх додатку (рис.24).

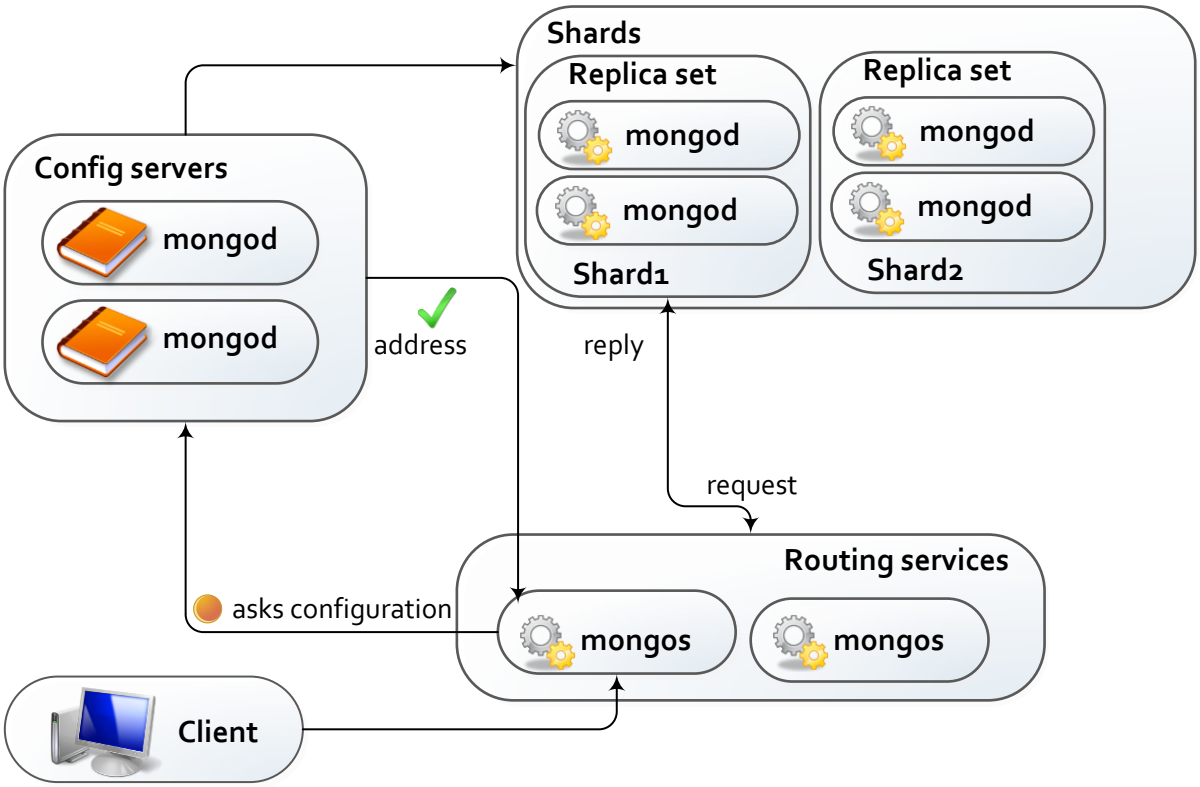



Рис.24. Принцип роботи шардінгу в MongoDB

В описанні процесу шардінга та на рис. 24 використовувалась наступна термінологія:

**client** - додаток, який використовує MongoDB;

**mongos (MongoDB shard)** - процес, який активований на одному або декількох серверах; та виконує функцію доступу до кластера (routing service);

**mongod** - процес, який відповідає за роботу з даними шарду, один шард може обслуговуватись декількома mongod-процесами, в тому числі розподіленими;

**config servers** - це процес(и), які зберігають метадані про шарди.

Розглянемо приклад. Припустимо у нас є три Debian сервери з адресами 127.0.0.1, 127.0.0.2 , 127.0.0.3.

Створюємо шард-сервери.

*127.0.0.2/путь_к_mongodb/mongod --shardsvr*

*127.0.0.3/путь_к_mongodb/mongod --shardsvr*

Отже у нас на серверах 127.0.0.2 та 127.0.0.3 працюють mongod процеси. Вони будуть обслуговувати наші шарди.

Створюємо config-сервери.

Створимо на сервері 127.0.0.1 сервер конфігурації нашого кластеру.

*127.0.0.1/путь_к_mongodb/mongod –shardsvr*

Створюємо mongos сервер.

*127.0.0.1/путь_к_mongodb/mongos --configdb 127.0.0.1:порт*

Так вказується серверу місце знаходження серверу конфігурацій, з якого він буде отримувати дані про шарди.

Конфігуруємо наш кластер.

*127.0.0.1/путь_к_mongodb/mongo*

*>use admin*

*admin*

*>db.runCommand( { addshard : "127.0.0.2:27018" } );*

*{"ok" : 1 , "added" : "shard0000"}*

*>db.runCommand( { addshard : "127.0.0.3:27018" } );*
*{"ok" : 1 , "added" : "shard0001"}*

Таким чином ми доставили серверу інформацію про шард-сервери в нашому кластері. Повідомимо сервер про технологію шардінгу.

*> db.runCommand( { enablesharding : "test" } );*

*> db.runCommand( { shardcollection : "test.users", key : {name:1} }).*

Таким чином сервер отримав повідомлення про необхідність здійснити шардінг колекції *users* із бази *test*, використовуючи для ділення на шарди ключ *name*. Важливо відзначити, що даний кластер готовий до шардінгу в межах визначеної колекції. Решта колекцій БД приймати участь в шардінгу не буде.

### 5.2.3. Особливості роботи з MongoDB

В системі MongoDB робота серверу конфігурується за допомогою параметрів, які знаходяться в файлі mongoDB.conf.

*Основними параметрами* є:

Dbpath - визначає шлях до місця зберігання файлів баз даних MongoDB;

logpath - шлях до файлу логування;

bind_ip, port - ip-адреса та порт з'єднання з MongoDB ;

nojournal - параметр, що визначає необхідність включення або відключення протоколювання (з використанням відповідного журналу) роботи MongoDB.

Приклад:

```
# store data in D:\MongoData
dbpath = C:\XLangHelp\xtools\tools\MongoDB\data
# log path
logpath = C:\XLangHelp\xtools\tools\MongoDB\logs\mongodb.log
# only accept local connections
bind_ip = 127.0.0.1
port = 27017
logappend = true
#journal = false
nojournal = true
#directoryperdb = true
# disable HTTP interface
# nohttpinterface = true
```

*Селектор запиту* MongoDB – поняття аналогічне параметру where SQL-запиту. Він використовується для пошуку, підрахунку, оновлення та видалення документів з колекцій. На відміну від SQL у якості селектору може використовуватися JSON-об'єкт, що надає обмеження на вибірку усіх документів. Наприклад, якщо ми маємо колекцію документів зі списком всіх студентів і нам необхідно вибрати студентів тільки жіночої статі, то при вибірці можливо просто використовувати селектор "{gender:'female'}".

Для створення індексу достатньо виконати в консолі MongoDB наступне:

```
db.stats_daily_domain.ensureIndex({"date":1, "domain" :1 } ,
{"name":"_date_domain"});
```

*{ "date" : 1, "domain" : 1 } – поля, за якими слід будувати індекс та його напрямок;*

*{ "name" : "_date_domain" } - назва індексу.*

*Мультиключі MongoDB* можуть автоматично створювати індекси для значень масивів. Теггінг – це приклад того, де це може бути корисно. Припустимо, що є об'єкт чи документ, який містить деякі назви, які є тегами:

```
obj = {
  name: "Apollo",
  text: "Some text about Apollo moon landings",
  tags: [ "moon", "apollo", "spaceflight" ]
}
```

та ці об'єкти зберігаються в db.articles. Команда :

*db.articles.ensureIndex( { tags: 1 } );*

проіндексує всі теги цього документа та створить індекси для "moon", "apollo" and "spaceflight" в цьому документі. Ми можемо звернутися до цих об'єктів звичайним способом:

```
> print(db.articles.findOne( { tags: "apollo" } ).name);
Apollo
```

БД створить індекс для кожного елементу масиву. Майте на увазі, що індексування об'єктів, які містять велику кількість елементів (сотні або тисячі) може бути ресурсозатратним.

### 5.2.4. Робота з MongoDB засобами .NET

Для роботи засобами мови C# необхідно використовувати драйвер https://github.com/samus/mongodb-csharp. Розглянемо відповідний приклад.

Основними просторами імен **MongoDB** є:

```
using MongoDB.BSON;
using MongoDB.Driver;
using MongoDB.Driver.Builders.
```

Для створення підключення до серверу використовується код:

*var connectionString = "mongodb://localhost/?safe=true";*

*var server = MongoServer.Create(connectionString).*

Якщо підключення пройшло успішно, то необхідно отримати об'єкт бази даних, з яким ми будемо працювати:

*var database = server.GetDatabase("test").*

Для додання даних можна створити, наприклад, два класи, які будуть описувати дані мешканців нашого під'їзду:

```
public class Flat
{
    public string FlatNumber {get;set;}
    public IList<Resident> Residents{get;set;}
}
Public class Resident {get; set;}
{
    public string Name {get; set;}
    public DateTime DateOfBirth {get;set;}
    public Email {get;set;}
}
```

В MongoDB ми постійно працюємо з колекцією документів. Одиницею є документ, який зберігається в БД в бінарному вигляді. Отримання посилання на колекцію, навіть якщо вона порожня, здійснюється командою:

*var collection = database.GetCollection<Flat>("flats").*

Далі вся робота проводиться з об'єктом collection. Створимо декілька об'єктів:

```
var flat1 = new Flat()
{
    FlatNumber = 1,
Residents = new List<Resident>()
    {
        new Resident ()
```

```
        {
                Name = "Шаповалов Дмитрий Анатольевич",
                DateOfBirth = Date.Time.Parse("1.12.1956"),
                Email = "shapa@domain.com"
        },
        new Resident()
        {
                Name = "Шаповалова Галина Ивановна",
                DateOfBirth  = DateTime.Parse("14.10.1995")
        }
    }
};
var flat2 = new Flat()
{
        FlatNumber = 1,
        Residents = new List<Resident >()
{
                new Resident()
        {
                Name = "Иванов Сергей Станиславович",
                DateOfBirth = DateTime.Parse("2.06.1989"),
                Email = " microsoft@domain.com"
}
    }
}
```

Додавання об'єктів до бази даних здійснюється командою insert.

```
collection.Insert(flat1);
collection.Insert(flat2).
```

# 6. Інтеграція корпоративних додатків

Інформатизація підрозділів підприємства сприяла появі складних програмних додатків, які були поступово розділені на незалежні, часто гетерогенні компоненти та підсистеми. Виникла проблема забезпечення можливості їх взаємодії для побудови цільної системи, що призвело до створення значної кількості програмних продуктів, які концентруються на інтеграції корпоративних додатків (Enterprise application integration - EAI).

До класу цих систем можна віднести: федеративні бази даних; системи орієнтовані на online-взаємодію (RPC, RMI та їх нащадки); системи, які орієнтовані на offline-взаємодію (message oriented middleware). Взаємодія базується на стандартизації протоколів обміну повідомленнями, стандартизації представлень даних, широкому застосовуванні мета-інформації.

## 6.1. Системи федеративних баз даних (Federated database systems)

Як правило, кожний відділ або департамент організації, має свої власні потреби в інформації і фокусується на конкретних додатках, пов'язаних з базою даних. Бази даних, додатки, мови і системи баз даних в різних департаментах являються однорідними в рамках департаментів, але є гетерогенними в межах організації – часто вони засновані на різних моделях даних, мовах, системах управління базами даних. Природно виникають задачі взаємодії усіх інформаційних систем організації (наприклад, для аналітичної обробки інформації). Одним з рішень є побудова системи управління федерацією баз даних (FDBS), що спрямована на забезпечення скоординованої роботи множини автономних гетерогенних баз даних. Вперше термін був запропонований у [Heimbigner, 85] для визначення взаємодії незалежних баз даних, але поряд з цим використовується абревіатура DACIS (Decentralized autonomus cooperating system). Як правило, FDBS не має власних даних, вона відповідає лише за координацію обробки запитів компонентами (системами, що включені до федерації). Головними задачами FDBS є: обмін даними між різними базами даних, консолідація ресурсів існуючого програмного забезпечення,

апаратних засобів і персоналу. Кожна база даних має свою власну автономію в плані, наприклад, обмеження її цілісності (integrity constraint), специфіки додатків, вимог безпеки. При цьому гетерогенність пов'язана не тільки з різними структурами даних, мовами запитів, управління транзакціями, але і з семантикою даних (схемою, призначенням атрибутів, зв'язків тощо, тобто їх інтерпретацією). Більш детально особливості таких систем, проблем та рішень представлені у роботах [Sheth, 1990], [Saltor, 1996], [Samos, 1998], [Ray, 2009].

Проблемами, що пов'язані з системами подібного класу, є наступні:

1. Вирішення питань, пов'язаних з відмінностями в моделі даних. Для цього може використовуватися єдина глобальна схема, що описує дані, або єдина мова, що описує обробку.
2. Відмінності в обмеженнях. Обмеження можливостей для специфікації і реалізації варіюватися від системи до системи, узгодження між якими здійснюються з використанням глобальної схеми.
3. Відмінності в мовах. Ті ж дані моделі, але з різними мовами можуть бути використані, їх версії можуть відрізнятися.

Проблеми семантичної гетерогенності мають місце, коли існують відмінності в розумінні, інтерпретації та скоординованому використанні даних з різноманітних джерел. При цьому виникають наступні задачі.

1. Автономність розробки. Відноситься до свободи вибору шаблонів проектування кожним із членів федерації: інформаційний домен для побудови схеми даних, представлення та іменування (representation and naming), семантика та інтерпретація даних (interpretation of data), обробка транзакцій та політика обмежень (transaction and policy constraints).
2. Автономність виконання (execution autonomy). Можливість формування порядку виконання.
3. Автономія зв'язку (communication autonomy). Відноситься до здатності вирішення щодо особливостей зв'язку з іншими компонентами.

4. Автономія асоціації. Можливість вирішити, якою функціональністю і в якому об’ємі можна забезпечити доступ до своїх ресурсів іншим компонентам федерації.

Для вирішення цих задач пропонується п’яті та восьми - рівнева архітектури систем.

### 6.1.1. П'яти-рівнева архітектура системи

В цій архітектурі використовуються три типи моделей даних: кожен компонент (автономна база даних) може мати свою рідну модель (native model); канонічна модель даних (canonical data model - CDM), яка прийнята в FDBS; зовнішні схеми (external schemas, import schemas), які можуть бути визначені в різних моделях користувальницьких додатків.

Крім того, використовуються три основних механізми для визначення схем на різних рівнях: переклад моделі (між CDM та іншими моделями), інтеграція схем (всі схеми визначаються за допомогою CDM), визначення похідних схем (*definition of derived schemas*).

Як показано на рис.25, кожен компонент бази даних має свою локальну схему. Локальні схеми перекладаються в схеми компонентів з застосуванням CDM. Проводиться узгодження для кожної бази даних: визначаються доступні дані, зі схем компонентів отримуються різноманітні схеми експорту, що визначають інформацію, яка може споживатися іншими компонентами. Це робиться за допомогою механізму визначення похідних схем. Кожна схема експорту є результат узгодження умов взаємодії (negotiation). Об'єднання схем експорту утворюють федеративні схеми (federated schemas). З кожної федеративної схеми можуть бути визначені зовнішні схеми, які визначають побажання до використання даних зовнішніх компонентів. Вони можуть бути сформовані з використанням механізмів отримання похідних схем та перекладу моделі.

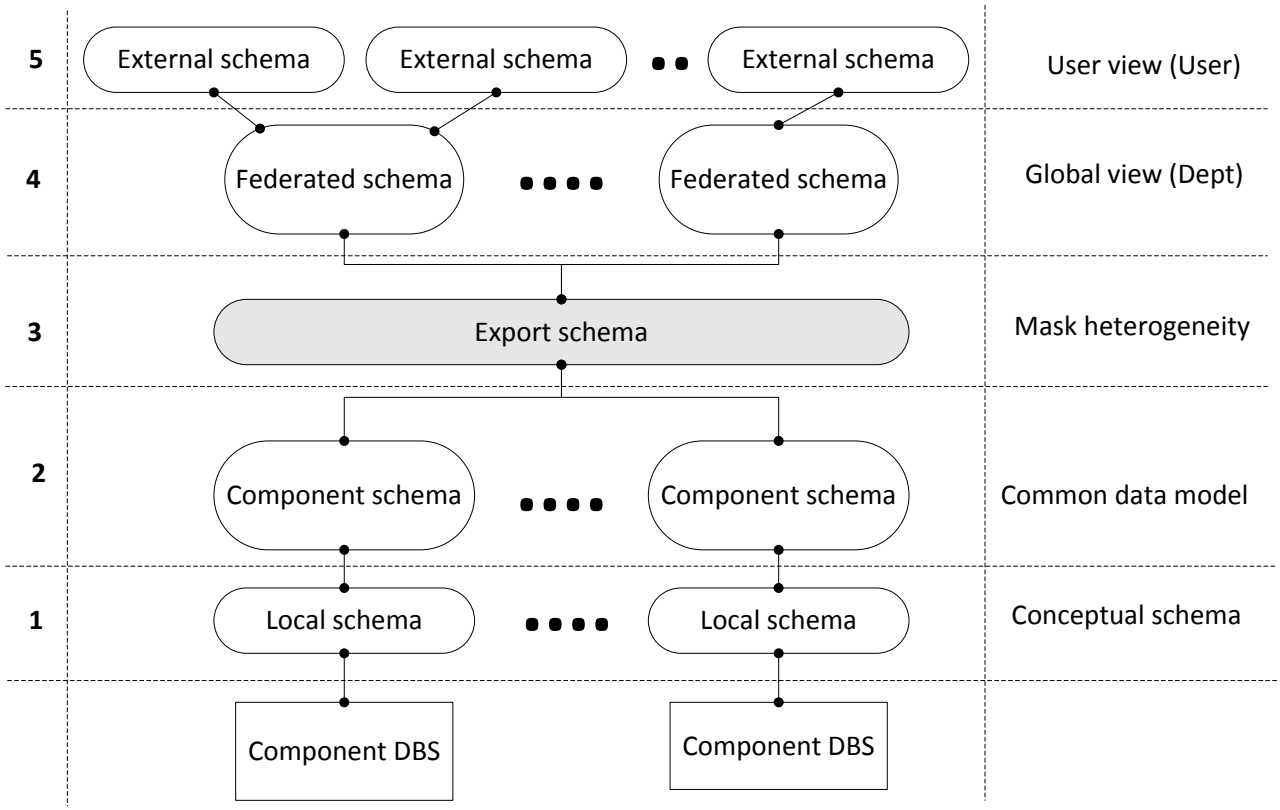



Рис.25. П'яти-рівнева архітектура

### 6.1.2. Восьми-рівнева архітектура

Восьми-рівнева архітектура [Saltor, 1996] є продовженням п'яти-рівневої архітектури з трьома новими рівнями схем, які стосуються трьох питань: визначення компоненту декількох федерацій; зовнішніх схем в користувальницьких моделях, які відрізняються від каноничних моделей; декілька значень рівня схеми об'єднання. При цьому визначаються наступні додаткові типи схем.

Схеми узгодження (*negotiable schema*) - визначаються в локальних схемах для кожного компоненту, використовуючи механізм отримання похідної схеми (тег 1). Як локальна, так і схема узгодження визначаються у вигляді локальної моделі (native model). Схема узгодження утворює схеми компонентів (тег 2). Локальні схеми можуть мати часткові елементи, які не мають бути експортовані в будь-яку федерацію.

Користувачеві зовнішні схеми (*user external schema*) визначаються в CDM з використанням механізму визначення похідної схеми (тег 6). Кожна зовнішня схема може бути перетворена для користувачів зовнішньої схеми в свою модель користувача, використовуючи механізм перетворення моделі (тег 7). В п'яти-рівневій архітектурі визначення зовнішніх схем і перетворення в моделі користувальницьких додатків робляться за один крок.

Схема додатку (*application schema*) - схема об'єднання, яка може підтримувати декілко семантичних складових, кожна з яких підходить для одного або декількох додатків. Таким чином, замість того, щоб зовнішні схеми безпосередньо визначалися зі схем об'єднання (federated schemas), вони визначаються зі схеми додатків (тег 6) за допомогою механізму визначення похідних схем.

Восьми-рівнева архітектура [Samos, 1998] показана на рис.26 .

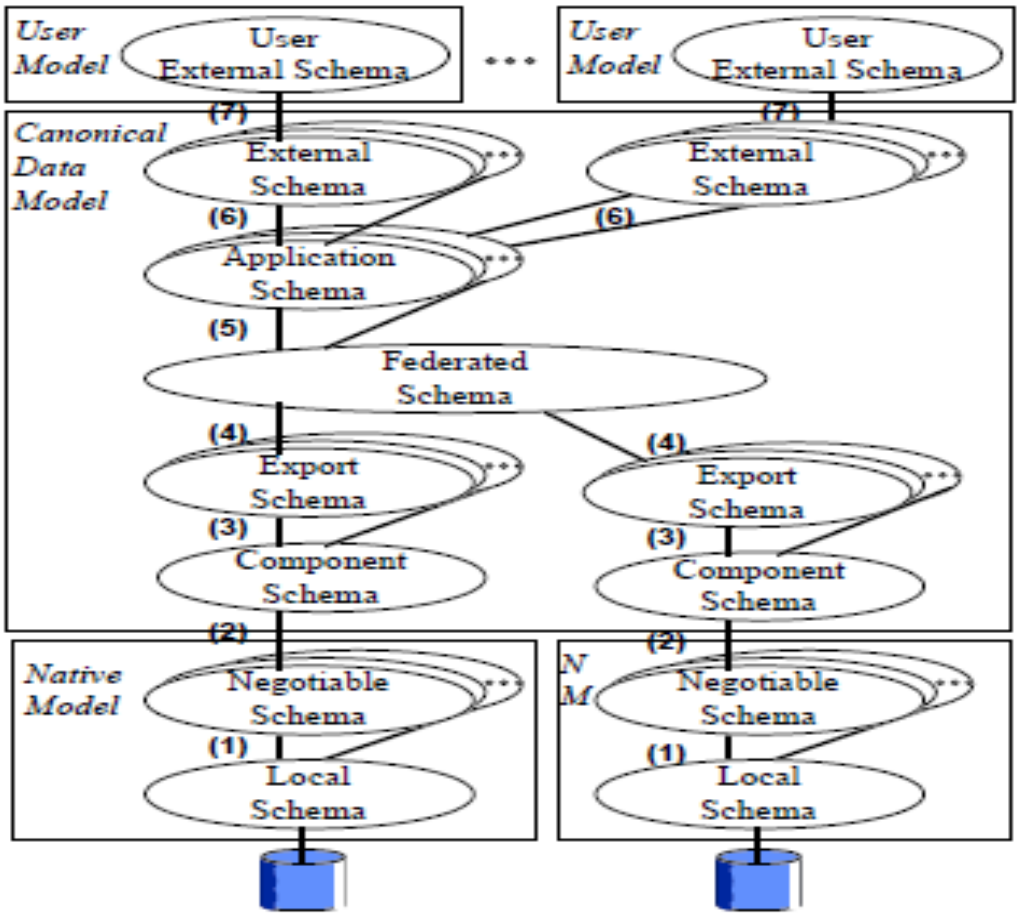



Рис.26. Восьми-рівнева архітектура [Samos, 1998]

## 6.2. Системи, орієнтовані на виклики віддалених процедур

Виклики віддалених процедур (RPC - Remote procedure calls) використовуються для реалізації функції доступу до компонентів - сервісів (network applications, services), розташованих на віддалених комп'ютерах. При цьому на відміну від звичайного механізму побудови взаємодії додатків шляхом пересилання повідомлень з відповідною інтерпретацією аргументів та результатів, та застосуванням API транспортного рівня (сокетів TCP), клієнт для взаємодії зі сервісом використовує виклик локальної процедури (local procedure call), яка приховує деталі відносно мережевої взаємодії, доступу до віддаленого компоненту, трансформації параметрів. По своїй функціональності система RPC займає проміжне місце між рівнем додатків і транспортним рівнем. У відповідності з моделлю OSI цьому положенню відповідають рівні уявлення і сеансу. Програмні реалізації системи, як правило, підтримують один або два протоколи. Наприклад, система RPC розробки фірми Sun Microsysttems (Open network computing - ONC RPC) підтримує передачу повідомлень з використанням протоколів TCP і UDP.

Ідея виклику віддалених процедур (Remote Procedure Call - RPC) полягає в поширенні механізму передачі управління і даних усередині програми, яка виконується на одній машині, на передачу управління і даних через мережу. Засоби віддаленого виклику процедур призначені для полегшення організації розподілених обчислень. Найбільша ефективність використання RPC досягається в додатках, де існує інтерактивний зв'язок між віддаленими компонентами з невеликим часом відповідей і відносно малою кількістю переданих даних. Такі програми називаються RPC-орієнтованими.

Процес виклику віддалених процедур. По-перше розглянемо виклик локальної процедури. Для здійснення виклику процедура заносить параметри в стек у зворотному порядку (рис.27). Після того, як виклик виконаний, процедура, що викликається, поміщає результат у регістр і повертає керування процеду-

рі, що здійснила виклик. Процедура вибирає параметри із стеку, повертаючи його в початковий стан.

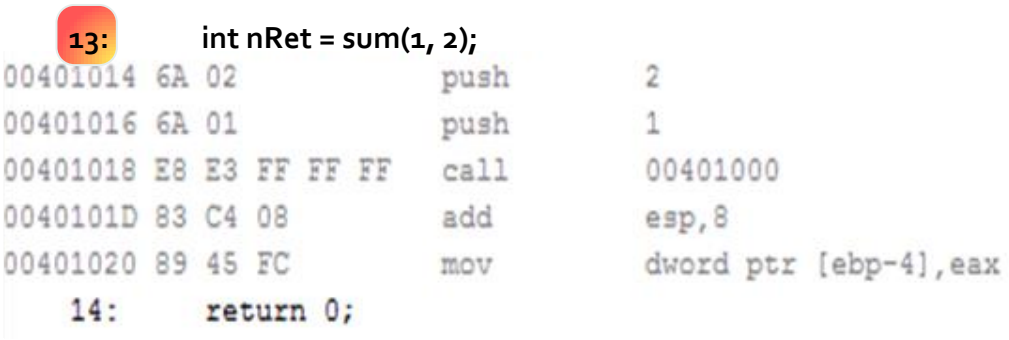



Рис.27. Виклик процедури.

Як правило, параметри можуть викликатися або за посиланням (by reference), або за значенням (by value). У разі передачі параметрів за значенням, для процедури, яка викликається, параметри-значення є локальними змінними, - вона може змінити їх і це не вплине на значення оригіналів цих змінних в процедурі, яка здійснила цей виклик. У разі передачі параметрів за посиланням (передається покажчик на змінну), варіація значення змінної призводить до варіації значення цієї змінної і в процедурі, що здійснила цей виклик.

Ідея RPC полягає в тому, щоб зробити виклик віддаленої процедури по можливості схожим на виклик локальної процедури. Це здійснюється наступним чином.

У разі виклику віддаленої процедури в бібліотеку поміщається замість локальної процедури інша версія, що має назву процедури-заглушки (stub). Вона виконує роль посередника у взаємодії з віддаленим сервером. Подібно оригінальній процедурі, стаб (stub) викликається з використанням визначеної послідовності. Тільки на відміну від оригінальної процедури він не поміщає параметри в регістри і не запрошує у ядрі дані. Замість цього він формує повідомлення до ядра віддаленої машини. Роль заглушки складається в отриманні необхідних параметрів (об'єктів) віддаленої процедури з адресного простору клієнта, перетворенні об’єктів у форму, яка відповідає стандарту

NDR (Network Data Representation) [NDR], здійсненні виклику служби RPC (бібліотеки), яка відповідає за виконання мережевого запиту. Мова XDR дозволяє описати складні формати даних в стислому вигляді. XDR мова схожа на мову С, але її призначенням є опис даних. XDR вписується в шарі презентації ISO (ISO presentation layer) і аналогічна за призначенням протоколу X.409 (абстрактна синтаксична нотація). Основною відмінністю є те, що XDR використовує неявну типізацію, а у X.409 використовується явна типізація. Упаковка аргументів і створення мережного запиту називається збіркою (marshalling).

Процедура - заглушка серверу очікує запит і при отриманні проводить розбірку (unmarshalling) об'єктів, які являються аргументами виклику процедури. Процес розбірки може включати також додаткові перетворення (наприклад, зміну порядку розташування байтів). Заглушка виконує виклик процедури-сервера, передаючи їй отримані розібрані об'єкти.

Після виконання процедури, управління повертається в заглушку сервера, передаючи їй результат обчислення. Як і заглушка клієнта, заглушка сервера перетворює повернені процедурою об'єкти, формуючи мережне повідомлення-відгук, який з використанням служби RPC та ряду інших служб рівня ОС передається по мережі до комп'ютера клієнту.

Операційна система передає отримане повідомлення службі RPC, яка, в свою чергу, передає його заглушці клієнта, яка вже в свою чергу, після розбірки, передає об'єкти процедурі-клієнту, дозволяючи отримувати результати віддаленої обробки так само, як нормальне їх повернення з процедури.

На цьому процес виклику віддаленої процедури завершено.

Табл.11 показує послідовність команд, яку необхідно виконати для кожного RPC-виклику, а рис.28 - ілюструє, яка частка загального часу виконання RPC витрачається на виконання кожного з 14 етапів (за даними приведеними в http://citforum.ru/operating_systems/sos/glava_12.shtml): виклик стабу, підготування буферу, упакування параметрів, заповнення поля заголовку, обчислення контрольної суми в повідомленні, переривання доступу до ядра,

черга пакету на виконання, передача повідомлення контролеру, передача по мережі Ethernet, отримання пакету від контролера, процедура обробки переривання, обчислення контрольної суми, перемикання контексту в простір користувача, виконання серверного стабу.

Послідовність команд шодо виконання кожного RPC-виклику

Таблиця 11

| | Клієнт | Сервер | |
|---|---|---|---|
| Клієнт | 1. Виклик стаб-процедури | 14. Обробка. | Сервер |
| Стаб клієнту | 2. Підготування буферу для відправки повідомлення. | Викликання серверу. | Стаб серверу |
| | 3. Упакування параметрів. | Поміщення параметрів у стек. | |
| | 4. Заповнення поля заголовка | Розпакування параметрів. | |
| | 5. Обчислення контрольної суми в повідомленні. | | |
| | 6. Переривання доступу до ядра. | | |
| Ядро | 7. Перемикання контексту простору ядра. Копіювання повідомлення в ядро. Визначення адреси призначення та поміщення в заголовок повідомлення. | 13. Перемикання на контекст серверу-стаба. Копіювання повідомлення до стабу сервера. Визначення стабу, призначення та його активності. | Ядро |
| | 8. Встановлення мережевого інтерфейсу. Включення таймеру. | 12. Обчислення контрольної суми, пере-вірка правильності пакету. | |
| | | 11. Обробка переривання після прийому пакету. | |
| | | 10. Прийом пакету. | |
| | 9. Передача по мережі Ethernet. | | |

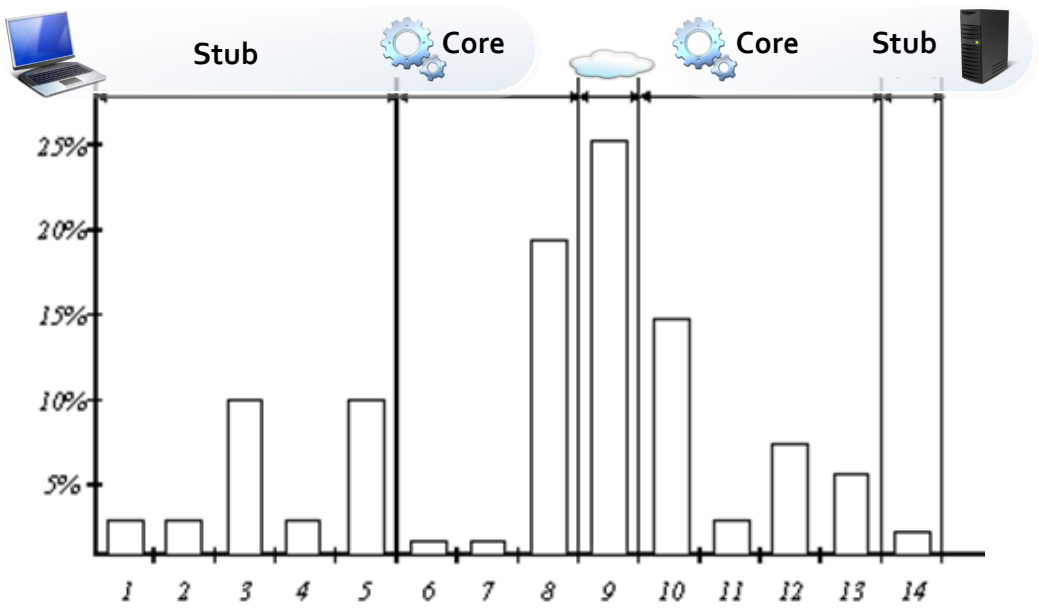



Рис.28. Розподіл часу між 14 етапами виконання RPC

Зв'язування (binding). Перш ніж клієнт зможе викликати віддалену процедуру, необхідно зв'язати його з віддаленою системою, котра володіє необхідним сервером. Таким чином, задача зв'язування розпадається на дві: знаходження віддаленого хоста з необхідним сервером, знаходження необхідного серверного процесу на даному хості.

Для знаходження хосту можуть використовуватися різні підходи. Можливий варіант - створення централізованого довідника, в якому хости анонсують свої сервери, і де клієнт може вибрати відповідні для нього хост і адресу процедури.

Кожна процедура RPC однозначно визначається номером програми (серверу процедур) і процедури. Номер програми - сервера визначає групу віддалених процедур, кожна з яких має власний номер. Кожній програмі-серверу також присвоюється номер версії, так що при внесенні в програму незначних змін (наприклад, при додаванні процедури) відсутня необхідність міняти її номер.

Таким чином, коли клієнт хоче викликати віддалену процедуру, йому необхідно знати номери програми, версії та процедури, що надає необхідний сервіс.

Для передачі запиту клієнту також необхідно знати мережеву адресу

хоста і номер порту, пов'язаний з програмою-сервером, що забезпечує потрібні процедури. Для цього використовується сервіс (демон), який надає сервіс віддалених процедур і використовує загальновідомий номер порту. При ініціалізації процесу-сервера він реєструє свої процедури і номери портів. Тепер, коли клієнту потрібно знати номер порту для виклику конкретної процедури, він посилає запит на сервер, який, у свою чергу, або повертає номер порту, або перенаправляє запит безпосередньо серверу віддаленої процедури і після її виконання повертає клієнту відгук. У будь-якому випадку, якщо необхідна процедура існує, клієнт отримує від серверу номер порту процедури, і подальші запити може робити вже безпосередньо на цей порт.

Семантика виклику. Виклик локальної процедури в загальному випадку призводить до її виконання, після чого управління повертається в головну програму. Але при виклику віддаленої процедури неможливо встановити, коли конкретно буде виконуватися процедура, чи буде вона виконана взагалі, а якщо буде, то яке число раз? Наприклад, якщо запит буде отриманий віддаленою системою після аварійного завершення програми серверу, процедура не буде виконана взагалі. Якщо клієнт при неотриманні відгуку після певного часу (тайм-ауту) повторно надсилає запит, то може створитися ситуація, коли відгук уже передається по мережі, а повторний запит знову приймається на обробку віддаленою процедурою. У такому випадку процедура буде виконана кілька разів, що може призвести до неправильних результатів.

Існують три базових стратегії виклику.

- Один і тільки один раз. Даної поведінки важко вимагати, зважаючи на можливі аварії серверу.

- Максимум раз. Така ситуація виникає при отриманні помилки замість нормального відгуку.

-Мінімум раз. В такій ситуації віддалена процедура повинна мати властивість ідемпотентності (від англ. idemponent). Цією властивістю володіє процедура, багаторазове виконання якої не викликає кумулятивних змін. Наприклад, читання файлу ідемпотентно, а додавання тексту в файл - ні.

Семантика відмов. Основними класами відмов є:

1. Клієнт не може визначити місцезнаходження серверу, наприклад, у випадку відмови потрібного серверу, або через те, що програма клієнту була скомпільована давно і використала стару версію інтерфейсу сервера. У цьому випадку у відповідь на запит клієнта надходить повідомлення, що містить код помилки.

2. Втрачено запит від клієнту до серверу. Рішення - через певний час повторити запит.

3.Втрачено відповідь - повідомлення від серверу клієнтові. Цей варіант складніше попереднього, так як деякі процедури не є ідемпотентними. На практиці використається послідовна нумерація всіх запитів. Ядро серверу запам'ятовує номер самого останнього запиту від кожного з клієнтів, і при отриманні запиту виконує аналіз - чи є цей запит первинним або повторним.

4. Аварія серверу після отримання запиту. Тут також важливу роль грає властивість ідемпотентності, але, на жаль, не може бути застосований підхід з нумерацією запитів. В даному випадку має значення, коли відбулася відмова - до або після виконання операції. Проте клієнтське ядро не може розпізнати ці ситуації, для нього відомо тільки те, що час відповіді минув. Основними підходами тут є: чекати до тих пір, поки сервер не перезавантажиться і намагатися виконати операцію знову; відразу повідомити додатком про помилку.

5. Аварія клієнту після відсилання запиту. У цьому випадку виконуються обчислення результатів, яких ніхто не очікує. Такі обчислення називають «сиротами». Наявність сиріт може викликати різні проблеми: непродуктивні витрати процесорного часу, блокування ресурсів. Існуючи рішення:

• Знищення. До того, як клієнтський стаб посилає RPC-повідомлення, він робить відмітку в журналі, сповіщаючи про те, що він буде зараз робити. Журнал зберігається на диску. Після аварії система перезавантажується, журнал аналізується і сироти ліквідуються. До недоліків такого підходу відносяться, по-перше, підвищені витрати, пов'язані із записом кожного RPC на

диск, по-друге, можлива неефективність через появу сиріт другого покоління, породжених RPC-викликами, поданими сиротами першого покоління.

• Перевтілення. Метод полягає у розподілі часу на послідовно пронумеровані інтервали. Коли клієнт перезавантажується, він передає широкомовне повідомлення всім машинам про початок нового інтервалу. Після прийому цього повідомлення усі віддалені обчислення ліквідуються.

• М'яке перевтілення аналогічно попередньому випадку, за винятком того, що відшукуються і знищуються не всі віддалені обчислення, а тільки обчислення клієнта, що перезавантажується.

• Закінчення строку. Кожному запиту відводиться стандартний відрізок часу T, протягом якого він повинен бути виконаний. Якщо запит не виконується за відведений час, то виділяється додатковий квант. Хоча це і вимагає додаткової роботи, але, якщо після аварії клієнту сервер чекає протягом інтервалу T до перезавантаження клієнту, то всі сироти обов'язково знищуються.

На практиці жоден з цих підходів не бажаний, більше того, знищення сиріт може погіршити ситуацію. Наприклад, нехай сирота заблокував один або більше файлів бази даних. Якщо сироту буде раптом знищено, то ці блокування залишаться і т.п.

Мова визначення інтерфейсу (*IDL).* У моделі RPC формально визначається інтерфейс віддаленої процедури, використовуючи для цього проектовану мову. Цю мову називають *мовою визначення інтерфейсу*, або IDL. Мова, реалізована фірмою Microsoft, у свою чергу називається мовою визначення інтерфейсу Microsoft, або MIDL.

Розглядаючи питання реалізації такого виклику, слід відзначити чотири основних компоненти RPC: компілятор IDL, виконуюча бібліотека і заголовок файлу, ім'я постачальника послуг (або локатор), картограф закінчення (або картограф порту). Виконуюча бібліотека складається з двох частин: бібліотека імпорту, яка пов'язана з додатком, і виконуюча бібліотека RPC, що реалізована як динамічно компонована бібліотека (dynamic link library - DLL). Після завершення створення інтерфейсу проводиться процедура компі-

ляції MIDL, в результаті якої генеруються заглушки, які призначені для перетворення викликів локальної процедури у виклики віддалених процедур. При цьому мережа стає повністю прозорою для розподіленого додатка.

У якості основних характерних рис виклику локальних процедур слід відзначити: асиметричність (одна з взаємодіючих сторін є ініціатором); синхронність (виконання призупиняється з моменту видачі запиту і відновлюється тільки після повернення з процедури).

### 6.2.1. Особливості технології RMI

У зв'язку з переходом розроблювачів прикладних програм від структурної парадигми до об'єктної з'явилася необхідність у використанні віддалених об'єктів (remote method invocation, RMI). Вилучений об'єкт − це дані, сукупність яких визначає його стан. Стан можна змінювати шляхом виклику його методів. Звичайно можливий прямий доступ до даних вилученого об'єкту, при цьому відбувається неявний вилучений виклик, необхідний для передачі значення поля даних об'єкту між процесами. Методи й поля об'єкту, які можуть використовуватися за допомогою вилучених викликів, доступні через деякий зовнішній інтерфейс класу об'єкта. Зовнішній інтерфейс компоненти розподіленої системи звичайно збігається із зовнішнім інтерфейсом одного із вхідних щодо компоненту класів.

В момент, коли клієнт починає використовувати вилучений об'єкт, на стороні клієнта створюється клієнтська заглушка, яку називають посередником (proxy). Посередник реалізує такий же інтерфейс, як й вилучений об'єкт. Породжений процес використовує методи посередника, який нормалізує їх параметри для передачі по мережі і передає їх по мережі серверу. Проміжне середовище на стороні серверу десеріалізує параметри й передає їх заглушці (її називають каркасом (skeleton)). Каркас зв'язується з деяким екземпляром вилученого об'єкту. Це може бути як знов створений, так і існуючий екземпляр об'єкту, залежно від застосовуваної моделі використання вилучених об'єктів.

Весь описаний вище процес називається маршалізацією вилученого об'єкту за посиланням (marshal by reference). На відміну від маршалізації за значенням, екземпляр об'єкту перебуває в процесі серверу і не залишає його, а для доступу до об'єкту клієнти використовують посередників. При маршалізації ж за значенням саме значення об'єкту серіалізується в набір байт для його передачі між процесами, після чого треба створити його копію в іншому процесі.

Виділяють три моделі використання вилучених об'єктів – модель єдиного виклику (singlecall), модель єдиного екземпляру (singleton), модель активації об'єктів щодо запитів клієнта (client activation).

При використанні даної моделі єдиного виклику об'єкт активується на час єдиного виклику. Для кожного виклику вилученого методу об'єкта клієнтом на сервері створюється й активується новий екземпляр об'єкту, який деактивується й потім вилучається відразу після завершення вилученого виклику методу об'єкта. Таким чином, вилучені виклики різних клієнтів є ізольованими один від одного. Завдяки видаленню об'єктів після виклику досягається економна витрата ресурсів пам'яті, але можуть витрачатися значні ресурси процесора на постійне створення й видалення об'єктів. Об'єкт не зберігає свій стан між викликами.

При використанні моделі єдиного екземпляру вилучений об'єкт існує не більше ніж в одному екземплярі.

При кожному створенні клієнтом посилання на вилучений об'єкт (точніше, на посередника) на сервері створюється новий об'єкт, що існує, поки клієнт не видалить посилання на посередника. При такому методі виклики різних клієнтів ізольовані один від одного, і кожний об'єкт зберігає свій стан між викликами, що призводить до найменш раціонального використання ресурсів пам'яті серверу. Це вирішується за допомогою механізму відстроченого виконання, який припускає збереження станів неактивних об'єктів на диску з наступним їх відновленням та виконанням.

### 6.2.2. Проблеми RPC систем

Зв'язаність компонентів (Coupling). Чиста модель RPC орієнтована на роботу з інтерфейсами об'єктів або функцій, в результаті чого компоненти через жорсткий механізм їх розподілу та зв’язку стають тісно пов'язаними (залежними один від другого) - будь-які зміни в інтерфейсі впливають на всю систему. В наслідок цього зміни в компонентах, або додавання нових компонентів збільшують вартість системи. Таким чином, використання чистої моделі RPCне є гнучким методом інтеграції декількох систем.

Надійність (Reliability). Надійний зв'язок може бути найважливішою характеристикою для розподілених додатків. Будь-який збій мережі, обладнання, послуг, іншого програмного забезпечення може вплинути на надійну передачу даних між системами. Більшість RPC реалізацій не надають гарантії надійного зв'язку, вони дуже уразливі до перебоїв в обслуговуванні.

Масштабованість (Scalability). У розподіленій системі, побудованій на базі RPC, блокуючий характер RPC може негативно вплинути на продуктивність в системах, в яких беруть участь підсистеми з різними рівнями масштабованості. Це уповільнює роботу системи до максимальної швидкості її самого повільного учасника. Синхронні взаємодії RPC потребують більшу пропускну здатність тому, що для підтримки синхронного виклику функції робляться додаткові запити (TCP).

Доступність (Availability). Системи, побудовані з використанням моделі RPC, є взаємозалежними, що вимагає одночасної наявності всіх підсистем; відмови в підсистемі можуть перевести всю систему до неробочого стану. В такий системі навіть тимчасова відсутність підсистеми у зв'язку з відключенням послуги або модернізацією може призвести до помилок в у всій системі.

## 6.3. Системи інтеграції розподілених додатків з використанням проміжного рівня

Системи, які базуються на застосуванні виклику віддалених процедур (RPC та RMI, та їх нащадків – DCOM, Web-services) мають недолік у тому, що викликаюча сторона та адресат повинні бути включені та працювати під час зв'язку. Крім того, вони повинні точно знати, як звертатися один до одного. Цей тісний зв'язок часто стає серйозною поміхою.

Спроби вирішити ці проблеми призвели до появи систем, що базуються на метафорі поштового сервісу – існує деякий проміжний шар програмного забезпечення (віддалених комунікаційних сервісів, розташованих на системах високої надійності), який відповідає за доставку повідомлень (message-oriented middleware – MOM) (рис.29, 30). Іншою метафорою такого слою є системи мережевої маршрутизації (routing system). Компоненти цієї системи просто відправляють повідомлення до логічних пунктів у мережі (точок доступу сервісів MOM), а інші компоненти отримують з MOM призначені їм повідомлення. При цьому вимоги щодо обов'язкової присутності компонентів взаємодії не ставляться, і це вирішує проблему доступності (reliability). Додатки можуть вказати свою зацікавленість до спеціальних типів повідомлень (зробити підписку на повідомлення такого типу), після чого шар MOM додатково подбає про те, щоб ці повідомлення були доставлені до тих додатків-передплатників. Ці системи типу публікація/підписка (publish/subscribe systems) формують важливий клас розподілених систем.

MOM системи забезпечують розподілені комунікації на основі моделі асинхронної взаємодії і це дозволяє MOM обійти багато обмежень RPC. Компонентам MOM-системи не потрібно чекати, доки повідомлення буде прийняте, вони дбають лише про відправку. Це дозволяє здійснити доставку повідомлень, коли відправник або одержувач не є активним або готовим відповісти. Але поряд з цим слід відзначити, що така доставка повідомлень, може зайняти у кращому випадку кілька хвилин (у разі відсутності клієнта на зв'язку - часів, або навіть днів) на відміну від механізмів таких як RPC (RMI),

для яких час виконання подібного завдання вимірюється в мілісекундах або секундах.

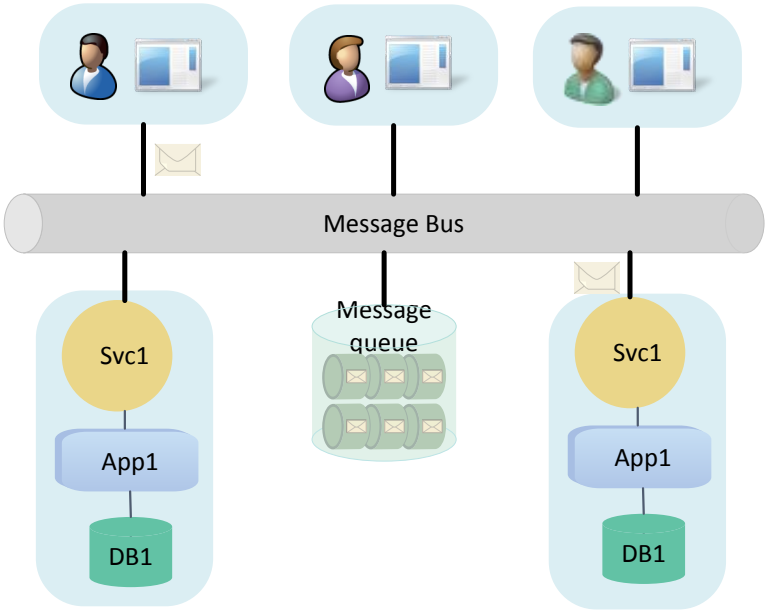



Рис.29. Використання програмного забезпечення MOM в якості посередника зв'язку при інтеграції корпоративних додатків

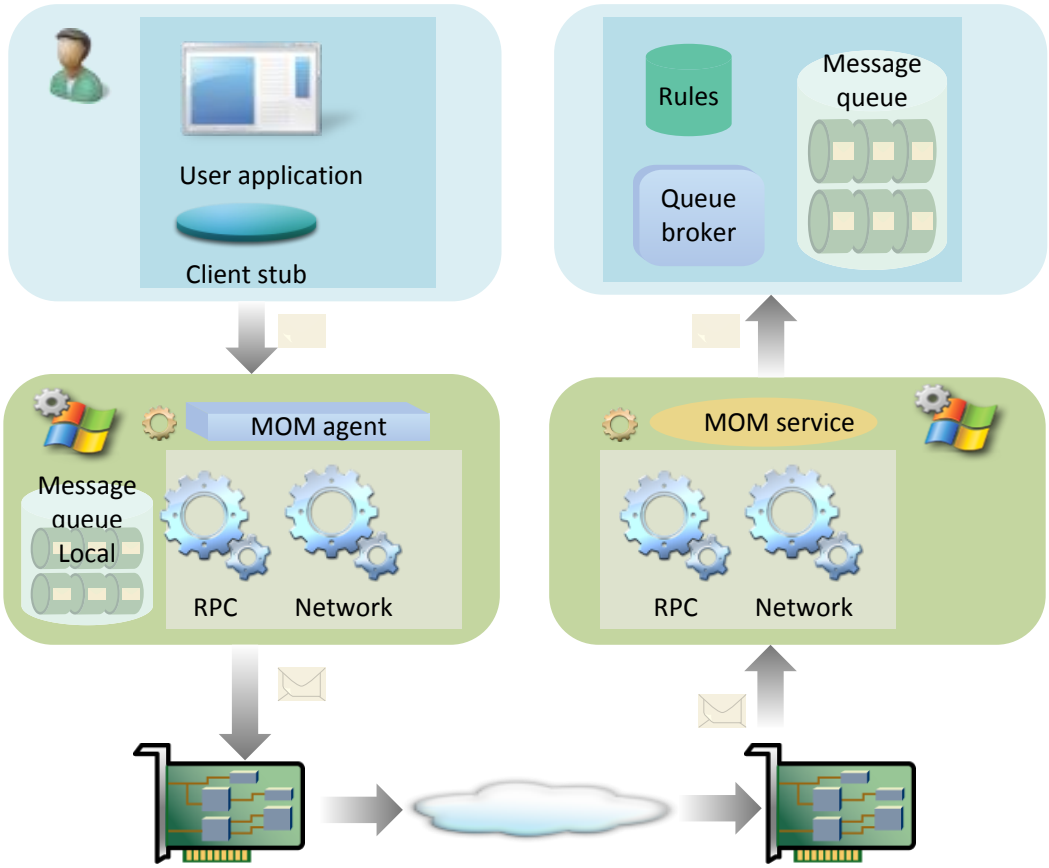



Рис.30. Практична реалізація інтеграції корпоративних додатків.

Крім того, при використанні МОМ відправка повідомлення не дає ніякої гарантії, що воно буде прочитано додатками, які перестали бути компонентами системи.Зв'язок (Coupling). Додатковий шар МОМ, який виступає в якості посередника для обміну повідомленнями, надає можливість уникнути тісного зв'язку між компонентами системи, що надає здатність зв'язувати додатки без необхідності адаптації один до одного – компоненти можуть навіть і не знати нічого один про одного, всі питання адаптації (семантичних деталей, пов’язаних з представленням інформації в повідомленнях) являються прерогативою шару-посередника.

Надійність (Reliability). Шар МОМ відповідає за захист системи від втрати повідомлень через помилки мережі або відмови системи, для цього застосовується механізм «зберігати і пересилати далі» (store and forward - SAF), за яким здійснюється гарантована пересилка повідомлень. У разі недоступності одержувача, відповідне повідомлення зберігається у черзі і пересилається при першому зручному випадку.

Масштабованість (Scalability). Розв'язуючи проблему тісної взаємодії підсистем, МОМ також вирішує проблеми залежності системи від характеристик підсистем. Підсистеми можуть масштабовуватися самостійно. МОМ також дозволяє системі справлятися з непередбаченими сплесками активності в одній підсистемі, розподіляючи навантаження між компонентами системи.

Доступність (Availability). Рівень МОМ відповідає за забезпечення високого рівню доступності системи, обробку перебоїв в роботі системи, дозволяючи користувачам працювати в безперервному режимі. Збій однієї з підсистем не впливає на роботу всій системи.

### 6.3.1. Черги повідомлень (Message Queues).

Черги повідомлень є базовим поняттям для систем МОМ. Черги забезпечують можливість зберігання повідомлень на платформах МОМ. МОМ клієнти мають можливість відправляти і отримувати повідомлення з черги. Черги займають центральне місце в реалізації моделі асинхронної взаємодії в

МОМ. Як правило, повідомлення, що містяться в черзі, сортуються в певному порядку. Стандартна черга має вигляд стеку First In –First Out (FIFO) - перші повідомлення, що надійшли, повинні бути витягнуті з черги першими (рис.31). Розширеним варіантом є FIFO-черга з пріоритезацією повідомлень.

Черга описується рядом атрибутів: ім'я, розмір, час життя (time to live - TTL), алгоритм сортування (message-sorting algorithm), інше. Використання черги повідомлень є особливо корисним для мобільних клієнтів без постійного підключення до мережі (наприклад, можливість відправляти замовлення в головний офіс торговим персоналом за допомогою мобільних мереж (GSM, GRPS і т.д.)). Клієнти можуть використовувати черги як тимчасову поштову скриньку, періодично перевіряючи, чи є для них нові повідомлення. Потенційно кожен додаток може мати свої власні черги, або додатки можуть використовувати одну загальну чергу. Як правило, МОМ - платформи підтримують кілька типів черг, кожна з яких має різні призначення. У таблиці 12 дається короткий опис типових черг МОМ.

Типові черги МОМ

Таблиця 12

| **Тип черги** | **Призначення** |
|---|---|
| *Public* | Відкритий доступ |
| *Private* | Приватний доступ. Підключення до черги вимагає від клієнтів аутентифікації. |
| *Temporary* | Черги створюються для певного періоду. |
| *Journal* | Черги призначені для ведення обліку повідомлень або подій, для цього копії повідомлень зберігаються в журналі повідомлень |
| *Connector / Bridge* | Включає власну реалізацію МОМ для взаємодії, наслідуючи ролі проксі для зовнішніх МОМ провайдерів. Міст (bridge) відповідає за переклад форматів повідомлень між різними постачальниками МОМ, що дозволяє клієнту одного постачальника мати доступ до черг / повідомлень іншого. |
| *Dead-Letter/ Dead-Message* | Черги повідомлень з вичерпаним терміном придатності або неможливості доставки (тобто, невірне ім'я черги або неможливость доставки за заданою адресою). |

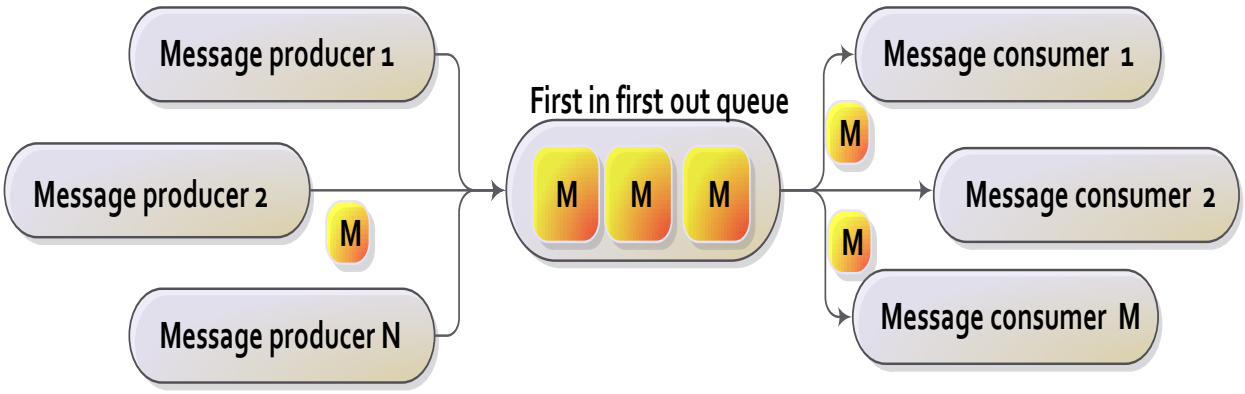



Рис.31. FIFO-черга повідомлень

### 6.3.2. Моделі передачі повідомлень (messaging models)

Існують дві основні моделі: точка-точка (point to point – P2P) і публікація/підписка (publisher/subscriber). Обидві моделі засновані на обміні повідомленнями через канал (черги). Типова система може використовувати гібрид цих моделей для досягнення різних цілей обміну повідомленнями.

Модель точка-точка (point-to-point) (рис.32) забезпечує простий асинхронний обмін, в якому повідомлення від клієнтів-постачальників направляються до клієнтів-споживачів, використовуючи чергу, при цьому кожне повідомлення буде доставлено тільки один раз, тільки одному споживачу. Модель дозволяє існування декількох споживачів визначеного повідомлення, підключених до черги, але отримати повідомлення зможе тільки один з них, після чого повідомлення буде видалене з черги. Призначенням такого механізму є надання можливості ефективного балансування навантаження в системі.

Модель публікації/підписки (publish/subscribe - pub/sub) використовується для розповсюдження інформації між анонімними споживачами і постачальниками повідомлень. Цей механізм дозволяє одному виробнику відправляти повідомлення сотням тисяч споживачів (рис.33).

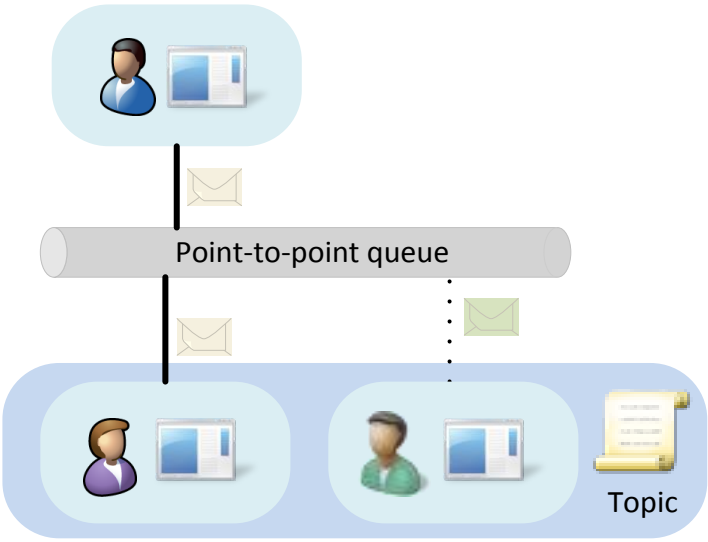



Рис.32. Модель обміну повідомленнями типу точка-точка

Додаток відправляє інформацію до рівня МОМ, що додає його відповідній черзі та здійснює відправку усім споживачам, що оформили підписку на повідомлення такого типу (topic). Наприклад, виробник-видавець може публікувати повідомлення про ціни на фондовому ринку або політичні події, ці повідомлення надсилаються до брокера-черги, який відправляє їх усім зареєстрованим абонентам-споживачам, що зареєструвались на дану тему (topic). В моделі не існує ніяких обмежень на ролі клієнта; клієнт може бути одночасно виробником і споживачем повідомлень заданого типу, пов'язаного з відповідною темою та каналом (чергою).

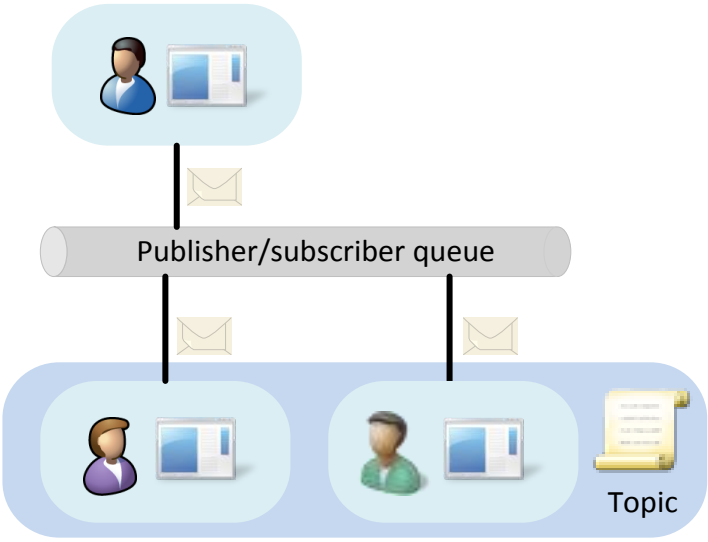



Рис.33. Модель обміну повідомленнями типу публікації/підписки

Існує ряд методів для публікації/підписки повідомлень, які підтримують різні функції, методи і алгоритми для фільтрації повідомлень, публікації, підписки та управлінням підпискою (табл.13).

Основні команди щодо публікації/підписки повідомлень

Таблиця 13

| Команда | Призначення |
|---|---|
| PUT (SEND) | Постановка повідомлення в чергу |
| GET (RECEIVE) | Прийом та видалення першого повідомлення, при цьому здійснюється блокування черги. |
| PEEK | Отримання копії повідомлення. Блокування черги не здійснюється. |
| PULL | Перевірка черги на повідомлення. Блокування черги не здійснюється. |
| NOTIFY (LISTENER) | Виклик додатку-обробника в момент надходження повідомлення до відповідної черги. Дозволяє інформувати клієнта про прибуття повідомлення за допомогою функції зворотного виклику на стороні клієнта. Функція зворотного виклику виконується при надходженні нового повідомлення у чергу |

### 6.3.3. Типові послуги MOM

При побудові великомасштабних систем важливо використовувати реалізації MOM, що орієнтовані на підприємство (enterprise level). На рівні підприємства такі платформи мають цілий ряд вбудованих послуг: виконання транзакцій, пошук сервісів, реплікація, узгодження протоколів, забезпечення надійної доставки, аутентифікації та управління доступом, балансування навантаження і кластеризація. Розглянемо основні послуги: фільтрацію та обробку транзакцій.

Механізм фільтрації повідомлень дозволяє спростити споживачеві здійснення вибору повідомлень, які він отримує від каналу. Фільтрація може працювати на декількох рівнях. Фільтри використовують логічні вирази, формат описання яких залежить від реалізації - зазвичай вони базуються на синтаксисі клаузи WHERE мови SQL-92. Моделі фільтрації зазвичай працюють

за властивостями повідомлення (пари ім'я/значення), але ряд проектів пропонують розширені моделі [CGI2]. Основні види фільтрації наведені в таблиці 14.

Перелік видів фільтрації

Таблиця 14

| **Тип фільтрації** | **Пояснення** |
|---|---|
| Topic/Channel - based | Події класифікуються в системі на декілька груп. Споживачі оформлюють підписку на групи за інтересами і отримують усі повідомлення, які знаходяться в групах. |
| Subject-based - based | В структуру повідомлень поряд з атрибутом, що вказує на канал, додається атрибут-тег, який надає інформацію про тему. Це надає можливість споживачам здійснювати фільтрацію типу «всі повідомлення з предметом, що починаються зі слів "Продаж автомобілів"». |
| Content-based | Дозволяє абонентам використовувати гнучкі мови запитів для завдання умов фільтрації щодо змісту цих повідомлень. |
| Content-based with Patterns (Composite Events) | Здійснює вибір повідомлень у разі надходження композиції подій [Pietzuch, 2003]. Наприклад, отримувати повідомлення від Майкрософт, з відповідною умовою на контент - тільки у разі існування повідомлень з визначеним змістом від компанії IBM. |

Транзакції. Важливою особливістю черг повідомлень є те, що вони можуть використовуватися як менеджери ресурсів для систем управління розподіленими транзакціями. Наприклад, система MSMQ тісно інтегрована з службою MTS, що надає можливість виконувати комплексні транзакції, компонентами яких можуть бути функції відправлення повідомлень.

Транзакції повідомлень використовуються у разі виконання декількох завдань, пов’язаних з доставкою одного або декількох повідомлень таким чином, що всі завдання будуть успішно виконані або всі відмінені. Повідомлення не пересилаються до кінцевого клієнту-споживача доки клієнт постачальник не підтверджує виконання транзакції (подібно до двох- фазового протоколу). Повідомлення зберігаються у черзі транзакцій, яка створена з

метою отримання та обробки повідомлень, що відправляються як частина транзакції. Слід відзначити, що нетранзакційні черги не можуть містити та обробляти повідомлення, які були включені в транзакцію. Деталі та особливості можуть бути знайдені в [Tai, 2003],[Liebig, 2001].

Механізми надійної доставки повідомлень. МОМ служби, як правило, дозволяють задавати конфігурацію якості обслуговування (QoS). При цьому задається режим доставки повідомлень: максимум один раз (at most once), хоча б один раз (at least once), тільки один раз і (once and once only).

Може також бути встановлений та налаштований режим підтвердження доставки повідомлень, задано число повторів у разі виникнення помилки доставки. У разі стійкої взаємодії (persistent communication) повідомлення зберігається, доки воно не буде доставлене визначеному споживачу (час зберігання визначається коефіцієнтом Time to live – TTL).

Для здійснення гарантованої доставки повідомлень (guaranteed message delivery) платформа зберігає повідомлення на дискі і потім відправляє повідомлення, чекаючи від споживача підтвердження доставки. Якщо підтвердження від споживача не надходить протягом визначеного часу - сервер відправляє повідомлення повторно і т. д. Цей механізм дозволяє відправнику повідомлення забути про проблему доставки повідомлення, покладаючись на МОМ. *Сертифікований метод доставки* (Certified message delivery) повідомлення є поширенням гарантований доставки - після отримання повідомлення споживачем, він формує підтвердження отримання (consumption report), що надсилається відправнику.

Формати повідомлень. В залежності від реалізації МОМ користувачеві можуть бути доступні ряду форматів повідомлень. Деякі з найбільш поширених типів повідомлень включає в себе текст (у тому числі XML), Object, Stream, HashMaps, потокове мультимедіа, і так далі. Наприклад, в системі MSMQ основною умовою до об’єктів є можливість їх серіалізації в XML.

Балансування навантаження. Існує два основні підходи до балансуванню навантаження: push (штовхати) і pull (витягувати).

В моделі push використовуються алгоритми для балансування навантаження між декількома серверами, суть яких полягає в визначені найменш навантаженого серверу і надсилання до нього запиту, при цьому використовуються методи прогнозування навантаження, що враховують конфігурації серверів та статистичні дані їх навантаження.

У моделі pull балансування навантаження створюється шляхом переміщення вхідних повідомлень у чергу point-to-point до якої підключено багато однотипних серверів. Ці сервери витягують повідомлення у своєму власному темпі. Це дозволяє забезпечити ідеальний механізм розподілу навантаження, так як сервер буде тільки тягнути повідомлення з черги, коли вони здатні її обробити.

Застосування методів кластеризації в системах МОМ дозволяє суттєво підвищувати рівень надійності системи, її доступність та ефективність роботи. Для цього використовують кластери високої доступності, розподілу навантаження. Наприклад, для забезпечення надійності системи стан сервера зберігається на декількох серверах і у разі відмови, це дозволяє клієнтові використовувати для виконання задачі альтернативний сервер, при цьому всі сервера можуть працювати у активному режимі, балансуючи при цьому навантаження.

Технології. У якості прикладів МОМ систем рівня підприємства можна назвати: IBM WebSphere MQ (MQSeries) [MQSeries], продукт Rendezvous компанії TIBCO [TIBCO], Microsoft MSMQ [MSMQ]. У кожного з цих продуктів свої власні API-інтерфейси і протоколи, також існують рішення для їх сумісного використання, узгодження стандартів. Особливо слід відзначити роль стандартного Java API орієнтованого на роботу з повідомленнями - Java Message Service (JMS) [JMS]. Java Message Service (JMS) визначає стандарт для надійного обміну повідомленнями підприємства. Завдяки поєднанню технології Java з МОМ системами, JMS API надає потужний інструмент для створення легко портуємих додатків, що написані на мові програмування Java. JMS API підвищує продуктивність праці програміста, визначаючи зага-

льний набір концепцій і стратегій програмування, який буде підтримуватися всіма JMS-сумісними MOM. Наприклад, для узгодження системи JMS з MSMQ можна використовувати open-source продукт MsmqToJMS (http://sourceforge.net/projects/msmqtojms/).

Поряд з основними технологіями слід відзначити ряд відкритих стандартів для надійного обміну повідомленнями: ebXML Service Message від OASIS [EBXML], протокол HTTPR компанії IBM[HTTPR], протокол WSReliability [WSReliability], протокол WS-ReliableMessaging [WS-ReliableMessaging].

## 7. Системна архітектура

При аналізі системної архітектури, як правило, виділяють два класи: централізований ( клієнт-сервер) і децентралізований (peer-to-peer або p2p).

Парадигма *клієнт-сервер* припускає чіткий розподіл між існуючими комп'ютерами: одні виконують обробку, зберігають дані, забезпечують керування, інші звертаються за необхідною обробкою. Парадигма p2p припускає розподіл рівних ролей всім елементам: кожний комп'ютер у мережі може бути і клієнтом і сервером і посередником передачі повідомлень. P2P системи дозволяють забезпечити підтримку надання послуг без використання дорогих серверів. І хоча основним напрямком є доступ до файлів (file sharing), вони з успіхом можуть застосовуватися для рішення різних завдань, пов'язаних зі зберіганням, пошуком і обробкою інформації. Так, P2P системи з успіхом використовуються у фінансових, оборонних, наукових, освітніх сферах. Основними напрямками є: використання значної кількості обчислювальних ресурсів для збільшення швидкості розрахунків, забезпечення надійності роботи системи в екстремальних умовах, розподіл збереженої інформації, забезпечення взаємодії різних модулів [Jia, 2005].

### 7.1. Клієнт-серверна модель системи

Розробка моделі «клієнт-сервер» була обумовлена дослідженнями в об-

ласті розробки операційних систем з метою забезпечення можливості використання користувальницькими додатками мережних ресурсів: здійснити спільне використання файлу, принтеру у формі доступної для користувача.

В 80-х роках комп'ютери керувалися монолітними операційними системами, у яких всі сервіси були її частиною. Для доступу до визначеної служби повна версія ОС повинна бути встановлена на комп'ютері, котрий надає дану послугу. Безумовною є необхідність виділення частини програмного забезпечення, що забезпечує дану послугу. Така частина одержала назву сервер, а кожний користувальницький процес - «клієнт», який може мати доступ і використовувати даний сервер. Таким чином, ідея побудови клієнт-серверної системи полягає в побудові частини ОС, відповідальної за забезпечення послугами користувальницьких машин як набору серверних процесів.

В 1990-і роки дана модель почала широко використовуватись для розробки великої кількості додатків. Така модель дозволяє програмістам прискорити розробку, тестування та інсталяцію додатків.

Відповідно до моделі існує два процеси: клієнт, що запитує послугу й сервер, який її надає. Сервер виконує запит і відсилає назад відповідь, це може бути результатом обчислень, підтвердженням завершення виконання операції або повідомленням про помилку в ході обчислень. Приклади сервісів: сервіс друку, файловий сервіс, сервіс аутентифікації, сервіс бази даних, сервіс певних обчислень. Всі сервіси пропонуються певними серверами.

Через те, що серверні ресурси обмежені, виникає конкуренція клієнтів за одержання послуги від сервера. У цьому випадку клієнтський і серверний процеси працюють на різних машинах і зв'язуються на віртуальному рівні шляхом обміну повідомлень. Логічний зв'язок процесів підтримується й фізичною передачею повідомлень між процесами.

Найбільш важливими особливостями клієнт-серверної моделі є простота, модульність, розширюваність і гнучкість. Простота проявляється в тісній відповідності потоків даних з контрольним потоком. Модульність досягається шляхом організації й інтеграції частини операцій в окремий сервіс. Крім

того, будь-який набір даних і операцій, виконаних над цими даними, може бути організований у вигляді окремого сервісу. Розподілена обчислювальна система, розроблена на базі клієнт-серверної моделі, може бути легко розширена за рахунок нових видів послуг, представлених у вигляді нових серверів. Сервери, які не задовольняють користувальницьким запитам, можуть бути легко модифіковані або навіть вилучені із системи. Єдина умова: інтерфейс між клієнтами й серверами повинен бути збережений.

Існує три основних проблеми клієнт-серверної моделі:

Перша − керування окремими ресурсами зосереджено на одному сервері. Якщо на комп'ютері, котрий підтримує сервер буде збій, то цей елемент керування стане недоступним, що неприпустимо у випадку, коли сервер відповідає за критичні системні функції, наприклад, сервер імен, файловий сервер, сервер аутентифікації та інше. Таким чином, надійність і доступність окремої операції, реалізованої сервером, залежить від сумарної надійності комп'ютерів, пристроїв і ліній комунікації даного сервера.

Друга – навантаження на сервер. Ситуація погіршується при збільшенні кількості клієнтських машин у мережі. Через мережу починає проходити занадто багато запитів, які навантажують сервер, при цьому обробка інформації може триватися недопустимо долго.

Третя проблема виникає, коли декілька реалізацій аналогічних функцій використовуються для підвищення ефективності роботи мережі через необхідність підтримки відповідностей у випадку застосування декількох однотипних серверів.

### 7.1.1. Основні шаблони клієнт-серверних систем

Для реалізації інформаційних клієнт-серверних систем існує багата кількість шаблонів: віддалені обчислення (remote computing); з використанням проксі-посередника (proxy computing); доставка коду (code shipping, code on demand); агентно-орієнтовані; чорна дошка (blackboard); публікація/підписка (publish/subscribe); відкрита агентна архітектура (open agent architecture).

Основний принцип, що лежить в основі клієнт-серверних обчислень - розподіл повноважень (separation of concerns). Правильне розділення функцій повинне спростити серверний компонент та поліпшити масштабованість системи. Базова модель не має чітких обмежень на розподіл функціональних особливостей, стану додатка між клієнтськими і серверними компонентами.

Базова модель віддалених обчислень (рис.34) полягає в відсутності будь-яких програмних компонентів та даних на стороні клієнта, клієнт має знати лише адресу сервера, протокол обміну повідомленнями, щоб на свої запити отримувати відповіді у стандартній формі, які є результатом обробки множиною сервісів-компонентів великих масивів даних.


Рис.34. Віддалені обчислення

Можна виділити два важливих різновиди таких моделей: зі станом обчислень (statefull) та без стану обчислень (stateless). Модель без стану (server-stateless) означає відсутність стану на стороні сервера. Кожен запит від клієнта до сервера повинен містити всі відомості, необхідні для розуміння запиту і не може покладатися на будь-який контекст, збережений на сервері, тобто стан сеансу залишається повністю на стороні клієнта. Це поліпшує властивості видимості, надійності і масштабованості системи. Видимість покращується, тому що система моніторингу не повинна виходити за рамки одного запиту для того, щоб визначити повний його характер. Підвищення надійності буде тому, що це полегшує процес відновлення після часткових відмов (partial failures). Масштабованість поліпшується по причині виключення обчислень зі стану, що дозволяє серверним компонентам швидко звільнювати

ресурси та спрощує балансування навантаження. Розширеня моделі (Client-Cache-Stateless-Server) складається у застосуванні кешу, який виступає в якості посередника між клієнтом і сервером, в якому відповіді на попередні запити можуть бути збережені та повторно використані для обробки подібних запитів. У якості варіанту stateful обчислень можна привести модель віддаленої сесії (remote session), яка намагається звести до мінімуму складності, або максимізувати повторне використання (reuse) клієнтських компонентів, перекладаючи контроль за сесіями на серверні компоненти. Клієнт ініціює сесію на сервері, викликає ряд сервісів на сервері, і по закінчені обчислень здійснює вихід з сесії.

Важливим компонентом систем мобільного коду є середовище виконання або хостінгу (computing environment), яке складає рівень віртуальної машини, або системи інтерпретатору, що відповідає за виконання програм в контрольованому середовищі, з належними рівнями безпеки та надійності. Це можуть бути і віртуальні машини скриптових мов, і середовища хостінгу вебсервісів. Особливістю цих систем є можливість динамічного перенесення процедур обчислень з одного вузла на інший, тобто міграції програмних компонентів завдяки середовищу виконання.

Віддалена оцінка (remote evaluation). За цією моделлю клієнтський компонент має знання, необхідні для виконання послуги, але не володіє ресурсами , які знаходиться на віддаленому сайті. Клієнт посилає компонент-оброблювач на віддалений сервер, який, в свою чергу, виконує код, використовуючи свої ресурси. Результати цього виконання потім відправляються назад клієнтові (рис.35). Такий підхід припускає, що відповідний код буде виконаний в захищеному середовищі (наприклад, що він не вплине на виконання завдань від інших клієнтів).

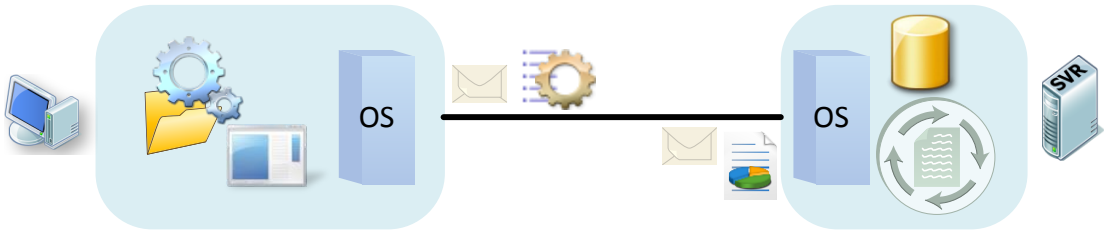



Рис.35. Віддалена оцінка

У моделі код за запитом (code on demand, code shipping) клієнтський компонент має доступ до ресурсів (даних), але не має засобів для виконання задачі (рис.36 ). Він посилає запит на віддалений сервер для отримання коду, який далі виконує локально. Переваги підходу включають в себе можливість додавання нових функцій в розгорнутому клієнті, що забезпечує поліпшену розширюваність і налаштованість (extensibility and configurability), також код може адаптувати свої дії до навколишнього середовища клієнтів і взаємодіяти з користувачами локально (offline). Простота зменшується через необхідність управляти моніторінгом навколишнього середовища. Масштабованість серверу покращується за рахунок винесення даних та обчислень на сторону клієнта. Найбільш істотним обмеженням моделі є складність моніторингу.

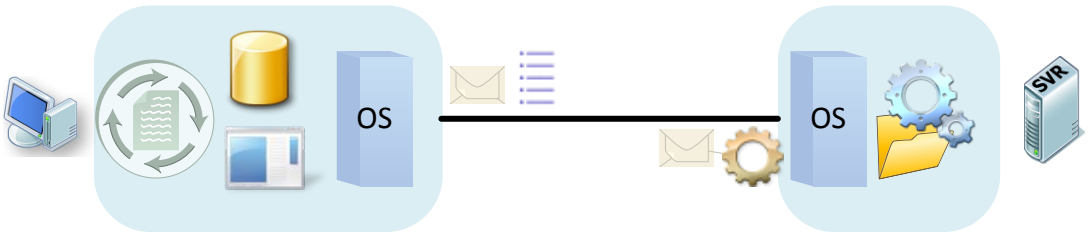



Рис.36 Код за запитом

Згідно з моделлю проксі-серверу обчислень, сервер відповідає лише за виконання обчислень, пов’язуючи дані та програмні компоненти, надіслані клієнтами, що володіють методами обчислень, але не мають відповідних потужностей для виконання обчислень (рис.37).

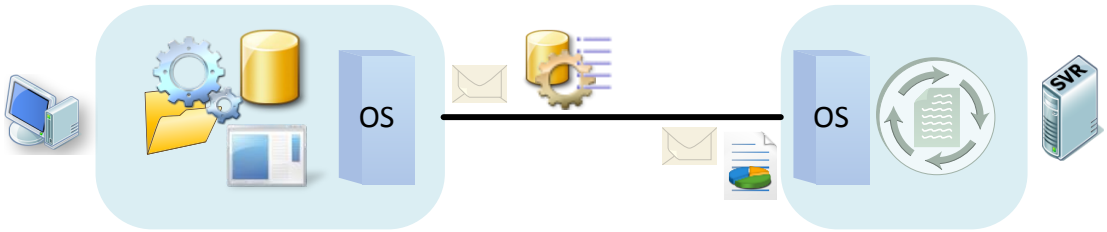



Рис. 37. Модель проксі - обчислень

### 7.1.2. Агентно-орієнтовані системи

Поняття агентно-орієнтованого програмного забезпечення є дуже широким. Наприклад, на сайті MIT (http://www.media.mit.edu/research/groups/software-agents) наведено таке визначення : «нова парадигма програмного забезпечення, яке діє як помічник користувача, забезпечуючи інтерактивний інтерфейс на відміну від просто інструменту, і який може навчатися взаємодії з користувачем, активно передбачуючи його потреби». Але , не зважаючи на функціональне різноманіття, основні характеристики можуть бути зведені до наступного переліку: проактивність (вміння передбачати ситуацію, а не тільки реагувати на неї), здатність до навчання з використанням попереднього досвіду, мобільність (можливість само-перенесення на інший комп'ютер), автономність, динамічна адаптація до умов що змінюються, активна взаємодія з середовищем виконання і іншими агентами (social ability), спрямованість на досягнення цілі (goal-orientation). Історію виникнення та розвитку парадигми детально викладено в роботі [Hyacinth, 96]. Спектр агентно-орієнтованих систем дуже широкий – наведемо лише основні типи таких систем.

Агенти орієнтовні на розмову/діалог/узгодження/адаптацію (conversational agents), широко застосовують технології обробки природної мови (natural language processing) для забезпечення текстового пошуку інформації та проблемно-орієнтованих діалогів. Існує багато типів таких агентів. Наприклад, агенти, що відповідають на запити клієнтів про продукти і послуги, або агенти, що спрямовані на допомогу студентам, надаючи поради щодо вирі-

шення проблем. У найближчі роки розмовні агенти будуть здатні підтримувати широкий спектр задач пов’язаних з бізнесом, освітою, державним управлінням, охороною здоров'я та розвагами.

Однією з причин цього є те, що розгортання автоматизованих рішень може значно знизити витрати на навчання і персонал. Використовуючи веб-технології, методи та технології комп'ютерної лінгвістики, розмовні агенти пропонують компаніям можливість надавати послуги клієнтам набагато більш економічно, ніж користуючись традиційними моделями. У варіантах безпосереднього спілкування (customer-facing deployment) агенти взаємодіють з клієнтами, допомагаючи їм отримати відповіді на свої запитання. У варіантах внутрішнього розгортання (internal-facing deployments) вони розмовляють з представниками служб обслуговування клієнтів, навчаючи їх допомагати клієнтам (рис.38).

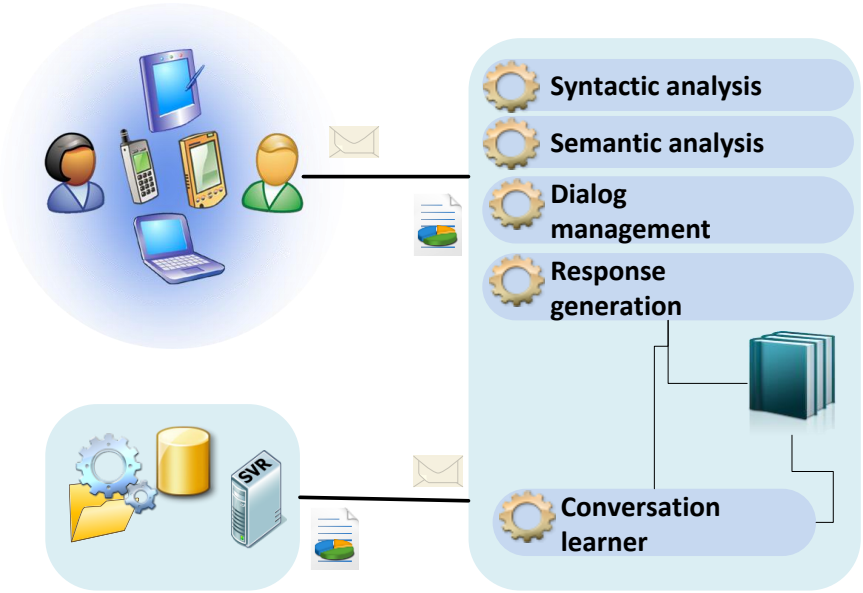



Рис.38. Агентна система орієнтована діалог

«Чорна дошка» (Blackboard) - це архітектура розподіленої обчислювальної системи, компонентами якої є інтелектуальні агенти (intelligent agents, knowledge sources, problem solving modules), які для взаємодії використовують спільний шар компонентів ("дошку"), що відповідає за зберігання черги завдань (database management subsystem, common global database) та відповідей, а також планування і управління процесом вирішення завдань (task

control subsystem). «Дошка» складається з двох компонентів: системи зберігання даних та системи управління завданнями, які взаємодіють з зовнішніми вузлами за допомогою системи передачі повідомлень (message passing kernel) (рис.39).

Шар компонентів (дошка) фізично може бути розташований на одному вузлі (централізована модель) або на декількох вузлах (розподілена модель). Термін «дошка» пов'язаний з метафорою обчислювальної моделі спільно працюючих навколо дошки фахівців. Для вирішення складної проблеми інтелектуальні агенти співпрацюють як фахівці, спостерігаючи за оновленнями інформації на дошці та реагуючи на ці зміни/події (event-driven process) шляхом самовизначення (self-actuating) завдань. Агенти оновлюють інформацію на дошці, записуючи повні або часткові рішення завдань, які споживаються іншими агентами у якості вхідних даних для вирішення наступних завдань і так далі. Загальний процес вирішення множини завдань відображається станом дошки (state of the blackboard, the specification of the original problem). Сучасний стан цієї архітектури розглянуто в роботах Daniel D. Corkill (http://dancorkill.home.comcast.net/~dancorkill/pubs/).

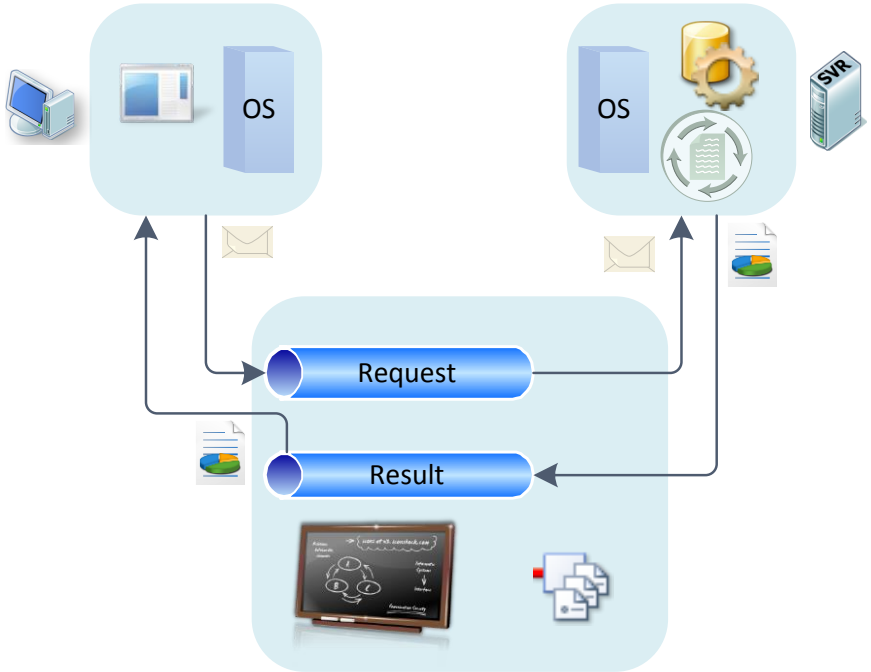



Рис.39. Архітектура «чорна дошка»

Стосовно цієї архітектури виділяють наступні переваги:

1. Інтеграція компонентів-клієнтів, що управляються централізованою системою керування.

2. Модульність (modularity) - кожен з компонентів-клієнтів є незалежним, що робить розробку і технічне обслуговування простіше.

3. Гнучкість (flexibility) – за рахунок слабкого зв'язку компонентів адаптація до змін вимог проходить набагато швидше ніж для традиційного процедурного програмного забезпечення додатків.

4. Розширюваність (extensibility) - нові компоненти можуть бути розроблені і застосовані в системі без зміни існуючої системи і без необхідності вказувати на своє існування в будь-якому іншому компоненті.

5. Ефективність і якість (efficiency and quality) – за виконання певної функції може відповідати декілька агентів, при цьому компонент управління системою може вибрати той, який забезпечить найбільшу продуктивність. Це може поліпшити як ефективність вирішення проблем, так і якість остаточного рішення.

6. Випадкова співпраця (opportunistic cooperation) – компоненти можуть направляти часткові рішення (partial solutions) до «дошки», інші - можуть використовувати ці часткові рішення для знайдення остаточного рішення, при цьому компоненти нічого не знають про існування один одного.

7. Повторне використання програмного забезпечення (software reuse) - незалежність і модульність компонентів надає можливість легкої побудови нових компонентів на базі вже існуючих. В якості компонентів-виконавців можуть бути застосовані традиційні процедурні модулі-додатки з відповідним рівнем адаптації до стандарту системи, що дозволяє уникнути додаткових витрат (investments) на побудову програмного забезпечення.

### 7.2. Однорангова модель системи

На противагу клієнт-серверній архітектурі найбільш сильні сторони P2P-базованих моделей засновані на їхній децентралізації й зменшенні зале-

жності від серверів, роль яких грають користувальницькі робочі станції. Деякі P2P моделі взагалі не потребують серверів. У цьому випадку користувачі можуть підключитися до інших користувачів без залучення серверу. Учасники обміну даними можуть контролювати правила з'єднання, які в клієнт-серверної версії їм недоступні . На відміну від клієнт-серверної моделі в P2P просто не існує критичного вузла, вихід з ладу якого призводить до відмови всієї мережі. У P2P-моделях з застосуванням серверів роль останніх зводиться до мінімуму. З урахуванням цих переваг багато корпорацій і комп'ютерних фірм вважають P2P модель такою ж важливою як і клієнт-серверну. Такі компанії як Intel і IBM витрачають зачні кошти на дослідження й розробки в області P2P.

# 8. Сервісно-орієнтована архітектура

## 8.1. Поняття сервісу та сервісно-орієнтованої архітектури

Вимоги багаторазового використання функціональних елементів інформаційних технологій, ліквідація дублювання функціональності в програмному забезпеченні, уніфікація типових операційних процесів привели до створення сервісно-орієнтованої архітектури (СОА). Ця архітектура представляє набір принципів, шаблонів, використання котрих гарантує досягнення поставленої мети. Сервісно-орієнтована архітектура забезпечує необхідну швидкість зміни програмного забезпечення у відповідь на зміну бізнес-правил, що лежать в основі життєдіяльності підприємства Така потреба найбільш гостро виникає на підприємствах із впровадженою системою керування на основі бізнес-процесів (Business Process Management Model), основною особливістю якої є тісна інтеграція - "бізнес - інформаційна система" [Aalst, 2002].

### 8.1.1. Підхід управління бізнес-процесами

У пошуках шляхів підвищення ефективності американського бізнесу на початку 1990-х років у США з'явилася нова парадигма організації бізнесу,

орієнтована на процеси (business-process oriented management).

Першим визначення бізнес-процесу (бізнес-процес або бізнес-метод) надав Davenport (1993), яке було подане у роботі «Process Innovation: Reengineering Work through Information Technology»: "Структурований, зважений (measured) набір активностей з виробництва певного продукту (specific output) для певного типу покупців. Основний фокус - яким чином здійснюється робота, на відміну від того, що виробляється. Процес - порядок просторово-часових дій, які мають початок і кінець, з визначенням входів і виходів, структури. Процеси - структури, якими організація робить необхідні дії для виробництва продукту, цінного для кінцевих клієнтів".

Інші визначення (в тому ж 1993 році) були дані Hammer та Champy, Rummler та Brache, Johansson. Найбільш повно про сутність, історію, розвиток принципу управління бізнес-процесами можна дізнатися у роботі [Brocke, 2010], в складанні якої приймали участь відомі фахівці в області бізнесу, інтеграції бізнесу та IT.

Згідно з Hammer [Brocke, 2010], ідея бізнес-процесу базується на двох складових: контролю якості, що базується на обробці статистичних даних, який був запропонований Walter Andrew Shewhart (1939) [Shewhart 86] та отримав розвиток у роботах у роботах William Edwards Deming [Deming, 86]; ідеї переробки існуючих процесів на підприємстві (business process reengineering) [Hammer, 90].

Підхід щодо контролю якості (quality approach) базується на статистичних методах управління (statistical process control), які спрямовані на ретельне вимірювання результату (outcome) із використанням статистичних методів з метою визначення та врегулювання причин, що приводять до виникнення проблем з продуктивністю.

Shewhart в 1939 році запропонував застосовувати науковий метод для виробництва продуктів або послуг на базі послідовних фаз «специфікація-виробництво-перевірка» (specification – production - inspection) (рис.40). У своїй роботі [Shewhart 86] він концентрує увагу на циклічності цих кроків:

«Ці три кроки повинні йти по колу, а не по прямій лінії… три кроки процесу виробництва можна порівняти з кроками в науковому методі. У цьому сенсі фази специфікації, виробництва та інспекції відповідають фазам генерації гіпотези, проведення експериментів і тестування гіпотез. Три етапи являють собою динамічний процес наукового пізнання».

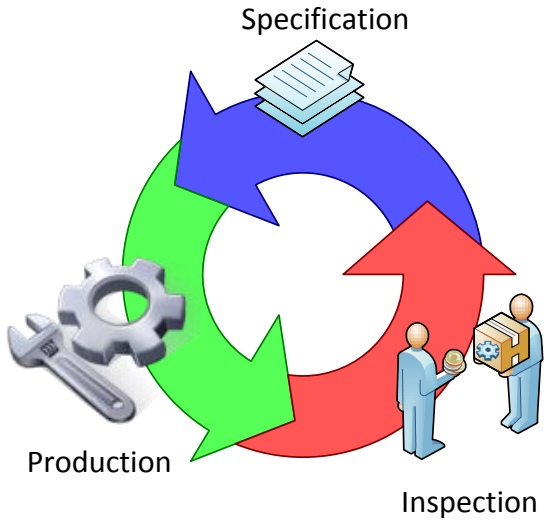



Рис.40. Три фази Shewhart.

Deming в 1950 році змінив 3-фазну модель «специфікація-виробництво-перевірка» на циклічну 4-фазну, додавши четверту фазу - тестування продукту за допомогою ринкових досліджень з подальшою його переробкою. Японська інтерпретація "колеса Демінга" (Deming wheel) призвела до появи принципу PDCA (plan-do-check-action). Цей цикл є невід'ємною частиною японської системи контролю якості QC, TQC. На рис.41 показані 4 фази: розробка продукту (з відповідними тестами); створення продукту з відповідною перевіркою якості; поставка до ринку; перевірка корисності продукту за допомогою ринкових досліджень – необхідно з'ясувати, що користувач думає про цей продукт, а також причини вибору споживачем іншого товару; повторна розробка продукту, з врахуванням реакції споживачів.

В 1993 році Демінг знову змінив цикл Shewhart і назвав його «цикл для навчання і вдосконалення» (plan–do–study–act (PDSA)), який розподілений на 4 фази: *планування* цілей (objectives), прогнозів (predictions), здійснення

подальшої роботи (carry out – «завдання – виконавець – умови – час»); *реалізація* з документуванням проблем та непередбачених ситуацій, аналіз даних по впровадженню; *аналіз* даних моніторингу, порівняння з прогнозами; *визначення заходів для покращення* ефективності роботи підприємства.

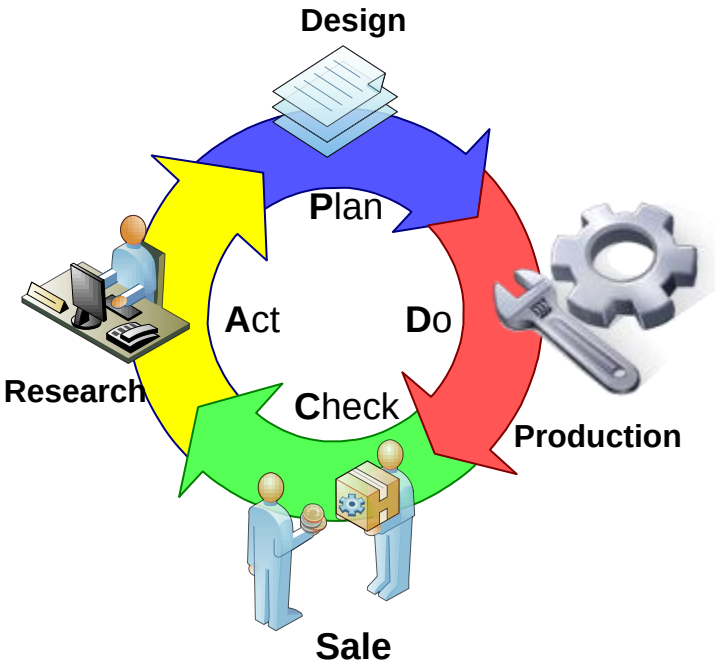



Рис.41. «Колесо Демінга». PDCA підхід.

Використання слова study для визначення третьої фази циклу підкреслює, що метою цього етапу є отримання нових знань. Але для побудови знань, слід вміти прогнозувати, чи призведе зміна до поліпшення в різних умовах, з якими доведеться зіткнутися в майбутньому. Для поліпшення моделі Langley, Moen та інші [Langley, 2009] запропонували додати до циклу PDSA три додаткові питання:

1. Чого ми намагаємося досягти?
2. Як ми дізнаємося, що зміна призведе до поліпшення виробництва?
3. Які зміни можна зробити для покращення виробництва?

Концептуальними принципами підходу до контролю якості є наступне: операції мають критичне значення і заслуговують серйозної уваги щодо визначення, планування, управління; використання метрик продуктивності, щоб визначити чи виконується робота задовільно; аналіз продуктивності ба-

зується на достовірних даних, а не на думках; джерелом проблем являються не люди, а об'єктивні причини; круг нескінченного вдосконалення (еволюційна модель), в якому рішення одних проблем, надає можливість переходу до постановки нових.

Якісний підхід страждає від двох обмежень. Перше – це визначення процесу як практично будь-якій послідовності операцій трудової діяльності. З цієї точки зору організація буде мати сотні або навіть тисячі процесів. Зосередження уваги на величезній кількості процесів вузької спеціалізації, які самі по собі не мають стратегічного значення для підприємства в цілому, заважає ясному розумінню ситуації та створенню відповідних рішень щодо управління. Друге - установка на ліквідацію варіацій і досягнення консистентної моделі роботи не дає можливості ефективного вироблення товару чи послуги. Процес може працювати без суперечень (consistent), без посилок, але все одно не досягти рівня продуктивності, необхідної для клієнтів і підприємств.

Двома основними концепціями, що були надані Michael Martin Hammer у роботах, присвячених переробці процесів, являються: поняття процесу, важливість розробки процесу.

Бізнес процес був визначений, як кінцева послідовність дій (end-to-end work), що існує в масштабах підприємства та створює цінність для клієнтів. Маючи справу з крос-функціональними процесами (розподіленими між декількома організаціями), вдалося визначити проблеми фрагментації: затримки; необґрунтовані витрати - що не впливають на цінність кінцевого продукту; помилки й складності, причина яких складається в тому, що робота виходить за рамки різних організацій, які мають різні пріоритети, різні джерела інформації , а також різні метрики.

Іншим важливим питанням стало визнання важливості розробки процесу, тобто визначення, яким чином його складові завдання зв'язані в єдине ціле (у підході управління якістю увага була сфокусована лише на виконанні та оцінюванні виконання процесів). Hammer визначив, що розробка процесу

(формалізація, моделювання) створює основу (envelope) для його виконання, що процес не може виконуватися краще, ніж дозволить його модель. І у разі, якщо вимоги до робочих характеристик перевершують те, на що здатна модель - модель повинна бути змінена на нову.

Націленість на результат і спрямованість на клієнта дозволяють уникнути ситуації з сервісами «розмазаними» по функціональних відділах компанії, у кожного з яких немає зацікавленості в кінцевому результаті. В результаті такої невизначеності повноважень падає якість послуг, замовлення обробляються не оптимально, з великими витратами: простої систем по причині поломок комп'ютерів; персонал не впроваджує навички, для яких створювались семінари та тренінги; отримання некоректних або пошкоджених даних та інш. У разі управління бізнес-процесами компанія використовує засоби розробки, моделювання та моніторингу для визнання і виправлення таких проблем.

Але на ефективність роботи підприємства впливає не тільки ведення бізнес-процесів, - споживачі, продукти та процеси формують залізний трикутник і, якщо організація не буде ставитися серйозно до однієї з складових, ефекту не буде.

Сумарне визначення бізнес-процесу може бути представлене у вигляді набору характеристик:

1. Definability - процес має чітко позначені границі, входи, виходи.
2. Order - процес включає зв'язані активності, впорядковані в просторово-тимчасовому контексті.
3. Customer - повинен бути споживач продукту процесу.
4. Value-adding - перетворення повинні являти цінність для клієнту.
5. Embeddedness - процес повинен бути вбудований в організаційну структуру.
6. Cross-functionality - як правило процес охоплює кілька функцій.

Особливістю реалізації процесного підходу є необхідність визначення об'єктів керування й детальної формалізації дій персоналу та систем з використанням ресурсів і чітких часових рамок для одержання кінцевого продукту або послуги (рис.42). Важливими складовими є визначення системи оцінки продуктивності для кожного з учасників процесу та застосування засобів моніторингу та оптимізації продуктивності.

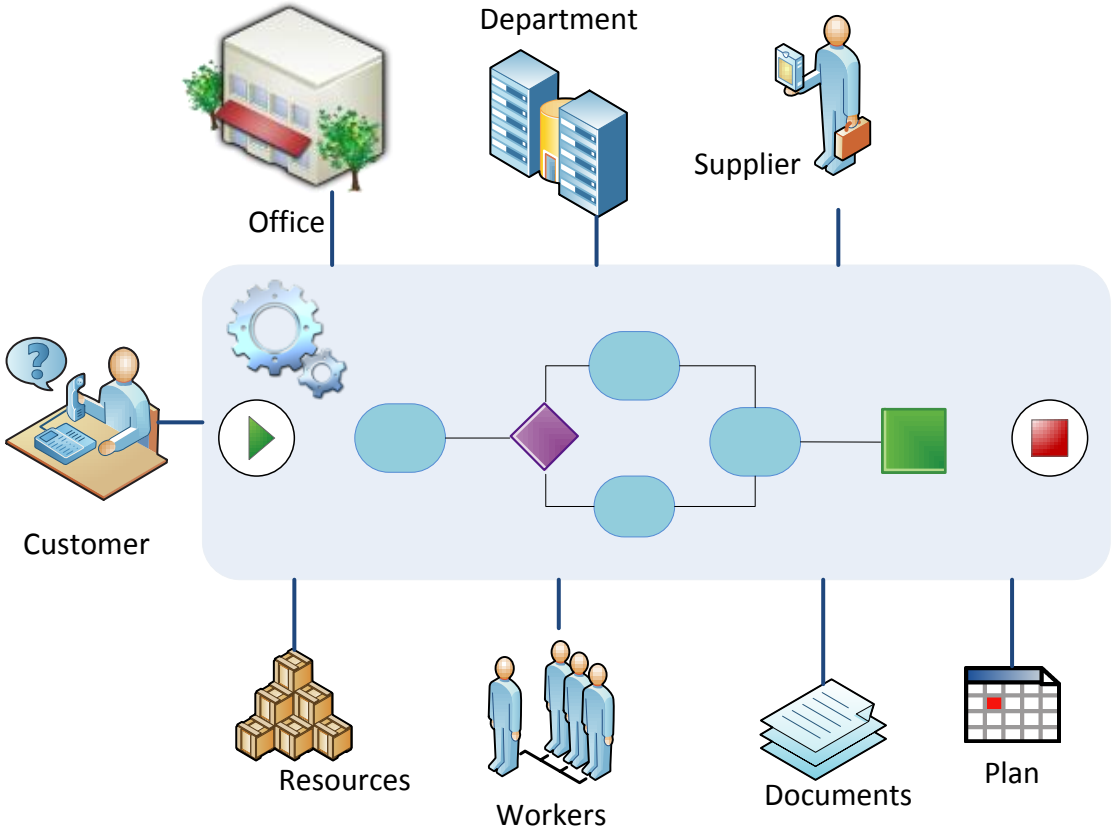



Рис.42. Загальний образ бізнес-процесу

Модель життєвого циклу бізнес-процесів являється похідною від моделі PDCA. На рис.43 показані стадії і їх взаємодія для досягнення безперервного вдосконалення певного бізнес-процесу.

На рис.44 приведена загальна модель управління бізнес-процесами. Виконавці процесу вимагають розуміння процесу в цілому і його цілей, вміння працювати в команді. Без цих характеристик вони будуть не в змозі реалізувати потенціал роботи. Важливою складовою процесу управління є визначення метрик – критеріїв оцінки ефективності процесу.

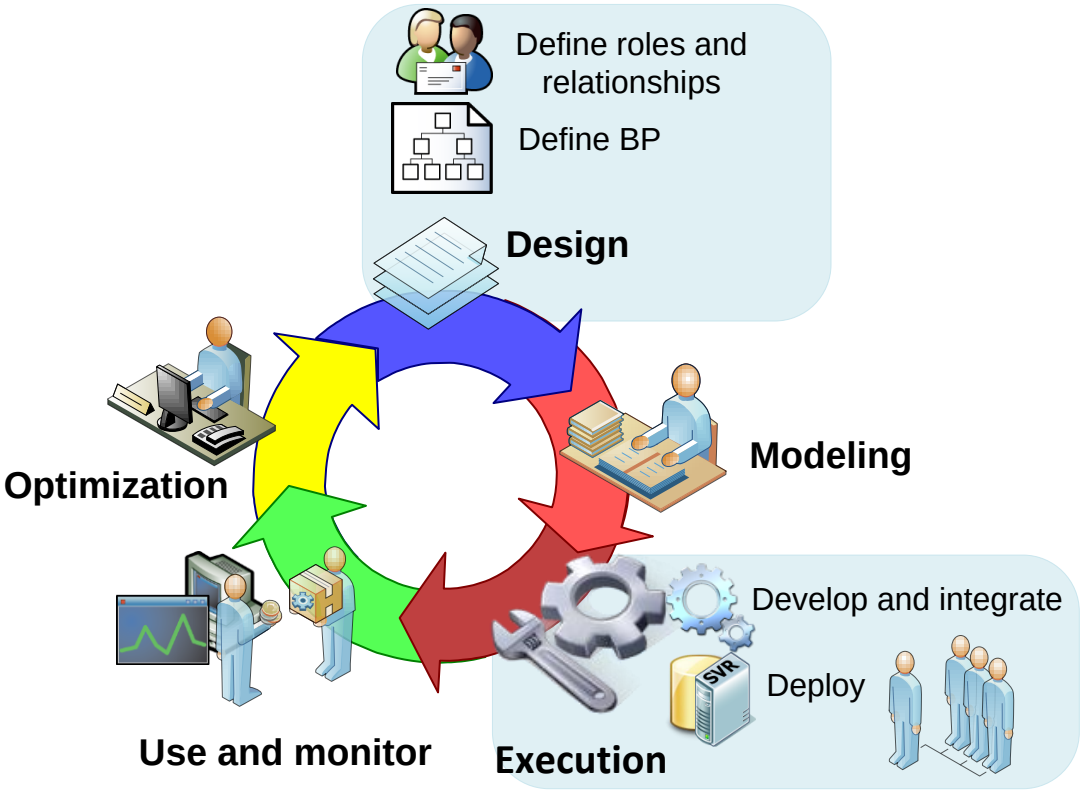



Рис.43. Життєвий цикл процесу

Більшість підприємств використовує функціональні показники ефективності, але це негативно впливає на роботу системи. Показники повинні бути похідними від потреб клієнту і цілей підприємства. Збалансований набір критеріїв оцінки процесу (таких, як вартість, швидкість і якість) застосовується таким чином, щоб поліпшення в одній області не приховували зниження та недоліки в іншому. Ще однією важливою складовою являється інфраструктура процесу. Виконавці потребують підтримки IT та HR (human resources) систем. Звичайні (функціонально фрагментовані) інформаційні системи не підтримують інтегровані процеси, а звичайні системи HR (навчання, компенсації, і кар'єра, і т.д.) спрямовані на відокремлені робочі позиції. Для інтегрованих процесів необхідні інтегровані системи (такі як ERP і системи компенсації, що орієнтовані на результат).

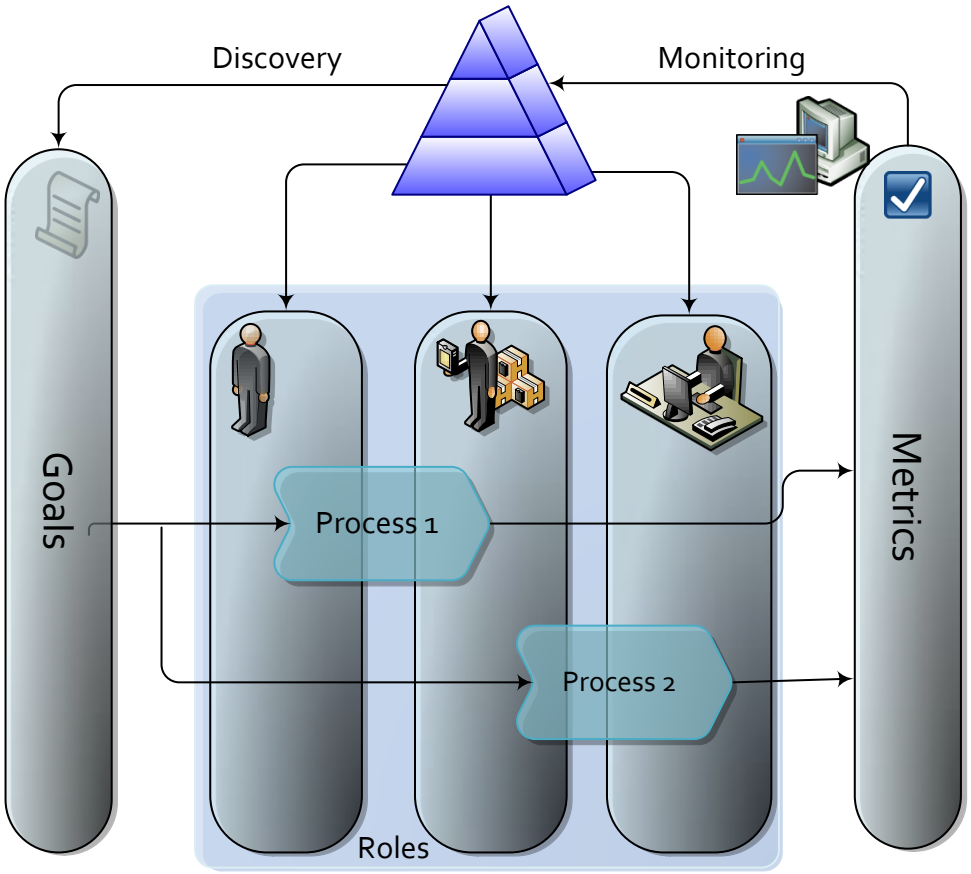



Рис.44. Загальна модель управління бізнес-процесами

### 8.1.2. Стадії бізнес - процесів

Стадія 1. Перша стадія життєвого циклу процесу - визначення ролей і зв'язків його учасників. У багатьох компаніях це робиться за замовчуванням ще до того, як використовується в інформаційних процесах. Ролі й зв'язки з іншими співробітниками визначаються для кожного співробітника, коли він тільки приходить у компанію. Великі компанії звичайно мають організаційні діаграми з ролями співробітників.

Стадія 2. Друга стадія життєвого циклу - визначення бізнес-процесів. Це визначення в загальному випадку виходить із необхідності автоматизації існуючих, виконуваних в ручному режимі процесів, або необхідності в новому процесі, пов'язаному зі змінами в стратегії компанії. Це стадія «паперового дизайну», на якій бізнес-процеси й вимоги до них визначаються на папері й документуються.

Стадія 3. Моделювання й оптимізація бізнес-процесів. Після того, як виявлені потреби й визначені вимоги, бізнес-аналітик переводить їх у серію завдань, які необхідно виконати для того, щоб задовольнити цим вимогам.

Це самий фундаментальний аспект моделі управління бізнес-процесами: специфікації, перелік, які завдання повинні бути виконані, ким, коли, в яких місцях, за яких обставин, в якій мірі точності, з якою інформаційною підтримкою тощо. Без розробки, специфікації процесу є тільки окремі неузгоджені види діяльності та організаційний хаос [Brocke, 2010]. Розробка здійснюється з застосуванням ряду формальних моделей. У цілому процес моделювання за J. Ownes (http://johnowensblog.com/) зводиться до наступних кроків: визначення каталогу бізнес-функцій (essential functions), створення на основі бізнес-функцій бізнес-процесів, створення процедурних моделей (зв'язок бізнес-процесів з розкладом), створення інформаційних моделей, створення моделей сутність-відношення, створення матриць залежності бізнес-процесів. Важливим поняттям являється бізнес-функція. Бізнес функція – це неділима активність, яка може приймати учать у різноманітних бізнес-процесах. Правильно побудована функціональна модель є визначення того, що бізнес повинен робити, незалежно від того, хто це робить і як це робиться. Функціональна модель повинна бути побудована першою тому, що вона являється основою для всіх інших.

Бізнес-аналітик може також змоделювати процес для того, щоб переконатися, що ця модель дозволить досягти необхідного результату. Такий порядок дозволить зробити припущення про обсяг завдань, вартість кожного кроку, наявність ресурсів і часу, необхідну для виконання кожного кроку. Ціль - створення повного визначення процесу з погляду бізнесу, щоб переконатися, що процес відповідає вимогам і очікуванням його власника.

Стадія 4. Розробка бізнес-процесу. На цій стадії бізнес-процес, описаний на попередньому етапі, перетворюється в практичне рішення, що може бути використано у реальній роботі підприємства: розробляються програмні компоненти, документи, формуються накази, інструкції. Для розробки про-

грамних компонентів застосовується спеціальні продукти – відповідний клас програмного забезпечення для управління бізнес-процесами. Технології можуть включати бази даних, системи управління повідомленнями, системи електронного документообігу, додатки масштабу підприємства, сервіси, інше. Далі проводиться тестування системи з широким використанням методів та засобів імітаційного моделювання.

Стадія 5. Інтеграція з іншими додатками. Після того, як процес розроблений, необхідно інтегрувати його з іншими додатками, вже існуючими на підприємстві. Для цього використовують різноманітні software development kits (SDKs), стандарт XML і веб-сервіси. У більшості випадків для інтеграції зі специфічними додатками (BPM-рішеннями) пропонуються додатки-посередники (proxy, adapters).

Стадія 6. Установка й адміністрування. Після того, як процес розроблено, протестовано, виконана інтеграція - він готовий до установки. Ця стадія вимагає контролю версій, міграції з однієї платформи на іншу й можливості конфігурувати й управляти BPM-системою та її різними компонентами.

Стадія 7. Виконання процесу. Ця стадія починається, коли бізнес-процес вже установлений. Учасники потоку робіт можуть починати використовувати й одержувати вигоду від BPM-системи. Користувачі працюють із процесом через користувальницький інтерфейс, що дозволяє їм виконувати різні функції й управляти своїми завданнями.

Стадія 8. Звіти. BPM-системи дозволяють одержати цінний зворотній зв'язок щодо ефективності і продуктивності всіх бізнес-процесів, а також продуктивності окремих учасників процесу.

Кожна стадія життєвого циклу вимагає залучення різних учасників, включаючи власників процесу, аналітиків, дизайнерів, розроблювачів, адміністраторів і кінцевих користувачів. Сучасне програмне забезпечення BPM пропонує відповідний інструментарій для підтримки всіх учасників на всіх стадіях життєвого циклу. Цей інструментарій повинен бути оптимізований для кожного типу учасників, що мають відношення до процесу. Таблиця 15

описує учасників і інструменти для кожної зі стадій. Узяті разом, ці інструменти становлять повну архітектуру BPM-рішення [Khan].

Життєвий цикл BPM

Таблиця 15

| Стадії | Інструментарій | Ролі |
|---|---|---|
| Визначення ролей і зв'язків | Організаційна діаграма | Власник процесу |
| Визначення процесу | Середовище моделювання | Власник процесу |
| Моделювання й оптимізація | Середовище моделювання | Бізнес-аналітик (business process analyst, chief process officer) |
| Розробка | Середовище розробки | IT-розроблювач |
| Інтеграція | Середовище розробки | Програміст |
| Інсталяція | Середовище адміністрування | IT-адміністратор |
| Виконання | Клієнтський додаток | Учасники процесу |
| Вимір | Звіти | Бізнес-аналітик |

### 8.1.2. Принципи СОА

Стратегія виконання даної вимоги (інтеграції "бізнес - інформаційна система") щодо сервісно-орієнтованої архітектури складається у використанні таких принципів як: стандартизація контрактів, слабка зв'язність компонентів (loosely coupling), абстракція, повторне використання, автономність, зменшення залежності логіки компонентів від стану й відстрочене виконання (statelessness), побудова модулів шляхом композиції існуючих сервісів (composibility), здатність до виявлення/знаходженя ресурсів (discoverability). При цьому очікуваним ефектом є: скорочення залежності між логічними підсистемами, компонентами, що беруть участь в автоматизації бізнес-процесів; можливість повторного використання одного компонента для різних цілей, процесів; масштабованість додатка; підвищення швидкості зміни й додавання нової функціональності.

Центральним поняттям СОА є поняття програмного сервісу [Kaplan, 2007], який можна визначити, як деякий слабо-зв'язаний, мета-орієнтований, у відмінності від завдання/функції, компонент, що володіє достатнім ступенем автономності (незалежності від користувача й інших підсистем). При цьому автономність компоненту розуміється, як приховування від користувача питань підтримки, установки, зміни компоненту, методів забезпечення його стабільної роботи. Користувач користується послугою, задаючи параметри й не замислюючись щодо особливості її якісного функціонування. Програмні сервіси є одним з підвидів програмних агентів і відповідно включають ряд їх характеристик: реактивність, асинхронність, автономність, наявність середовища хостинга, мобільність [Martin, 1996].

Сервіс може бути також представлений як набір функціональних можливостей-послуг (capabilities), що надається компонентом. При цьому виділяють дві складові: логіка-реалізація, що становить тіло сервісу, й фасад-контракт (edge), що дозволяє забезпечити його використання.

З сервісом також пов’язане поняття точки доступу (end point) , що визначає логічну адресу (URI - Uniform Resource Identifier), за якою користувач може знайти контракт. Логічна адреса далі відповідними службами транслюється у IP-адресу, порт, ім’я сервісу, композиція яких ідентифікує додаток. Робота з сервісами базується на передачі повідомлень з застосуванням стандартних протоколів (HTTP, SOAP). Структура сервісу наведена на рис. 45.

Зв'язок основних принципів показаний на рис.46. Основою СОА являється стандартизація контрактів. Стандартизація базується в першу чергу на стандартах описання політик, функцій та типів (рис.47), але до цього принципу відносять також і структурованість та логічну цільність сервісів – уникнення порушень принципу розподілу повноважень (separation of concerns).

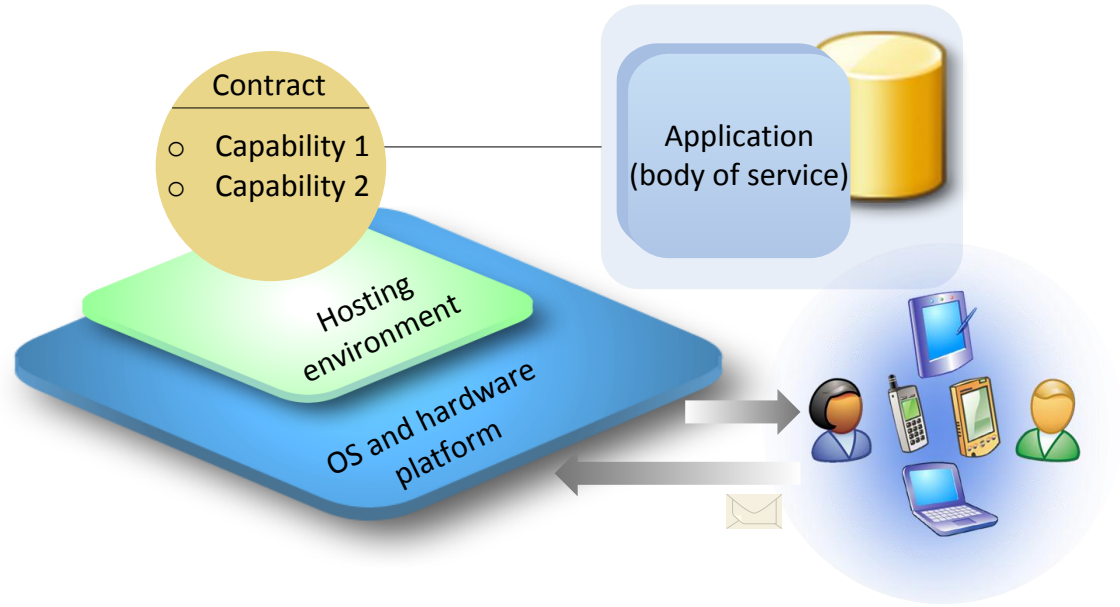



Рис.45. Структура сервісу

Кожний сервіс є цінним IT-активом (valuable asset), який відповідає стратегічним цілям підприємства, тому до розробки сервісів підходять серйозно. Важливим для СОА є початок побудови системи з визначенням контрактів (contract first) – контракти з'являються до реалізації компонентів додатків. Контракти же визначаються на базі специфікацій -моделей бізнес-процесів .

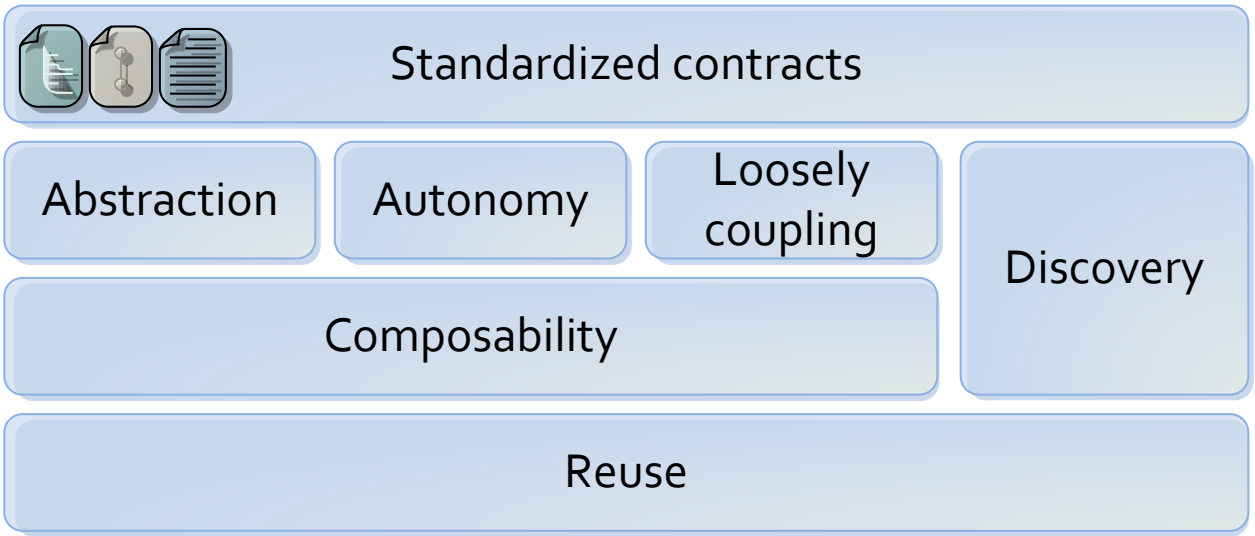



Рис.46. Зв'язок основних принципів СОА.

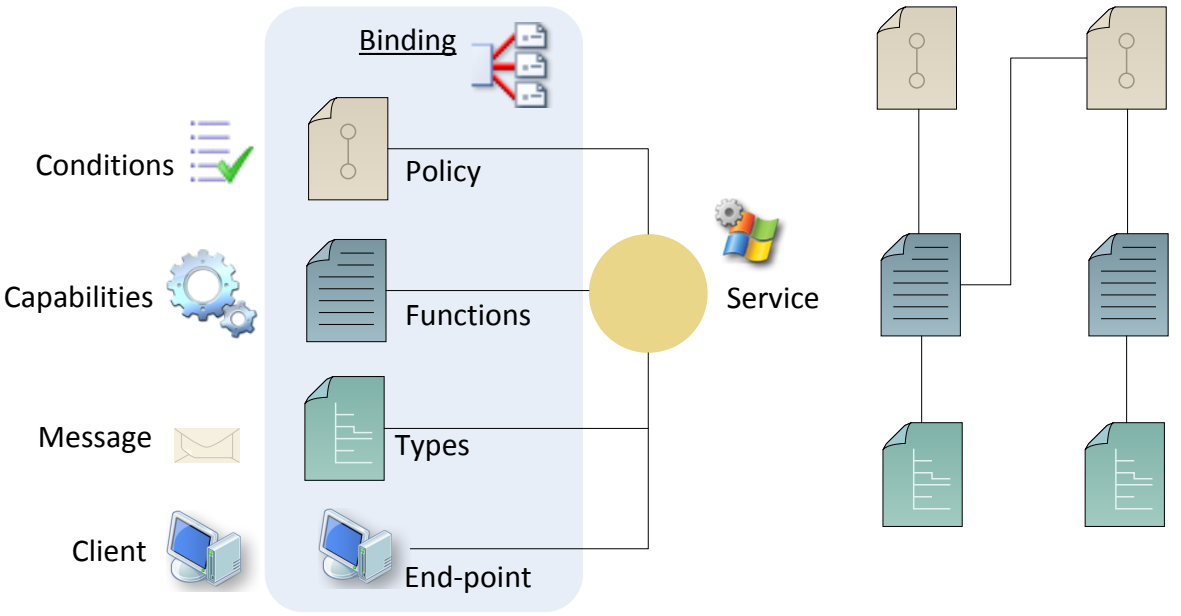



Рис.47. Принцип стандартизації контрактів

Принцип абстрактності сервісів полягає у створенні агностичних інероперабельних сервісів, які не тільки не залежать від платформи та засобів реалізації, але й від змін бізнес правил, процесів. Важливою складовою абстракції є приховування деталей реалізації логіки сервісу, - так за оболонкою сервісу може стояти застаріле ПЗ (legacy software), множина компонентів, пов'язаних з базою даних або ієрархія сервісів. Це забезпечує поступовий перехід підприємства до СОА, адаптуючи вже існуючі підсистеми до інтегрованого середовища.

Основною ціллю СОА є створення деякого реєстру автономних нормалізованих сервісів (service inventory), що відповідають бізнес-функціям (essential business processes), і формування на базі композиції елементарних сервісів-компонентів служб, відповідальних за автоматизацію діючих бізнес-процесів (рис.48). Виділення таких автономних сервісів – непроста задача, вирішення якої залежить від нормалізації бізнес-функцій, повноти описання бізнес-процесів, вірного формування контрактів. Тому існує ствердження, що нормалізація сервісів не відповідає принципам гнучких методів розробки (counter-agile) і цілком залежить від розробника. Важливими принципами при цьому являються розподілення відповідальності (SR – single responsibility) та

виділення інтерфейсів (IS – interface segregation). Автономність сервісів – основа повторного використання компонентів, легкої адаптації ПЗ до змін, підвищення якості, збереження коштів на розробку та підтримку ПЗ (визначаються коефіцієнтом повертання інвестицій – ROI return of investments). При цьому слід відзначити, що використання нормалізованих сервісів залежить від здатності їх находження у реєстрі на замову розробника або користувача. Такий пошук здійснюється за функціональними можливостями, класом діяльності, призначенням, розробником та іншими властивостями, які мають бути наданні з поставкою сервісу. Описання сервісів складають базу знань. За управління даною базою відповідає система, яка може бути доступна користувачу через локальний або публічний сервіс. Вона відповідає за класифікацію і винахід (discovery) необхідного сервісу або групи сервісів у відповідь на замовлення.

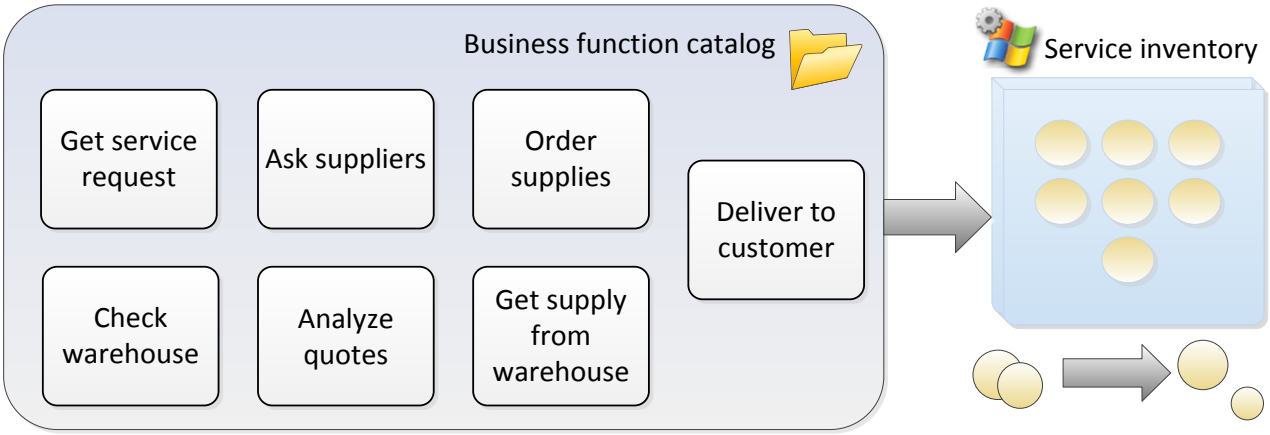



Рис.48. Принцип нормалізації сервісів

Принцип щодо можливості створення композиції (composability) базується на принципах автономності та абстрактності. У зв'язку із цим розрізняють наступні види сервісів: сервіси-сутності, сервіси-завдання й сервіси-функції. Сервіс-сутність (*entity service*) – це сервіс, що забезпечує роботу (functional boundary) з бізнес-сутностями, такими як пацієнт, клієнт, службовець інше. Даний тип сервісу відрізняється високим ступенем повторного використання, тому що він не залежить від конкретного бізнес-процесу, але розділяється ним. Сервіс-завдання (*task service*) – це сервіс, пов'язаний з ви-

конанням певного завдання або процесу. Він має меншу здатність повторного використання і застосовується як контролер композиції, що відповідає за композицію інших більш агностичних сервісів (process-agnostic services). Сервіс-утиліта (*utility service*) – це сервіс, призначений для підтримки функціональності системи й не зв'язаний безпосередньо з бізнес-контекстом системи. Сервіс-утиліти становлять шар технологічно-орієнтованих сервісів (technology-oriented service layer).

Особливе місце займають сервіси-агрегати: диригування (orchestration) та хореографії (choreography) (рис.49 ). Сервіси диригування базуються на формальному, імперативному описі послідовності дій та умов, за якими ці дії виконуються. У якості дій, як правило, використовується взаємодія з сервісами завдань. Сервіси диригування складають платформу автоматизації бізнес-процесів, пов'язуючи сервіси бізнес-функцій у відповідному порядку. Хореографія – декларативне формальне описання координації множини учасників із зазначенням їх ролей і спостереженням за обміном повідомленнями.

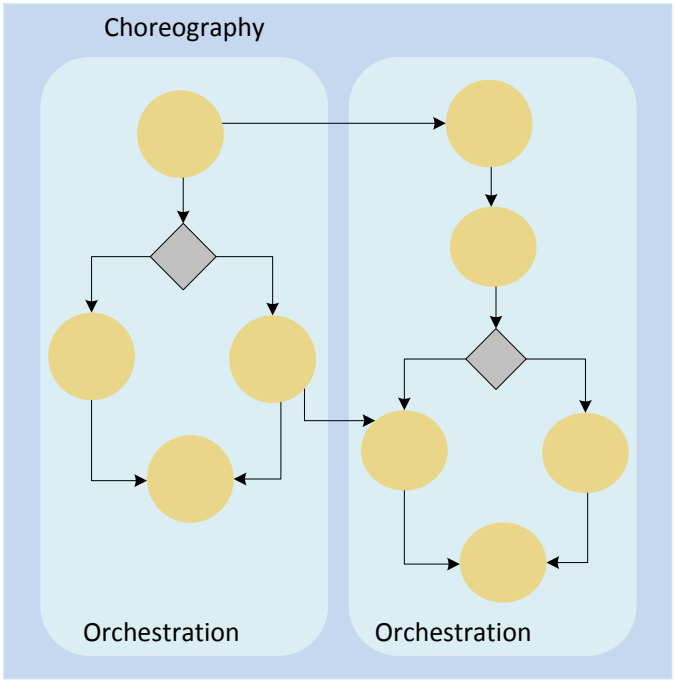



Рис.49. Сервіси диригування та хореографії

Програмні комплекси, розроблені відповідно до СОА, часто реалізуються як набір інтегрованих за допомогою відомих стандартних протоколів

(SOAP, WSDL, і т.п.) сервісів. Інтерфейс компонентів СОА-програми дозволяє ізолювати деталі реалізації конкретного компоненту (ОС, платформи, мови програмування, постачальника та інше) від інших компонентів. Таким чином, СОА надає гнучкий спосіб комбінування й багаторазового використання компонентів для побудови складних розподілених програмних комплексів.

### 8.2. Web-сервіси

Web-сервіси застосовуються для обміну даними між сервером і клієнтом (рис. 50) - забезпечують мережевий доступ до функціональності програмного додатку на базі стандартних Інтернет-технологій з використанням протоколів HTTP, SMTP, XML-RPC, SOAP. Слід відзначити незалежність від платформи (операційної системи, середовища розташуван-ня) та засобів реалізації (мови програмування) [Tidwell, 2001],[Cerami, 2002].

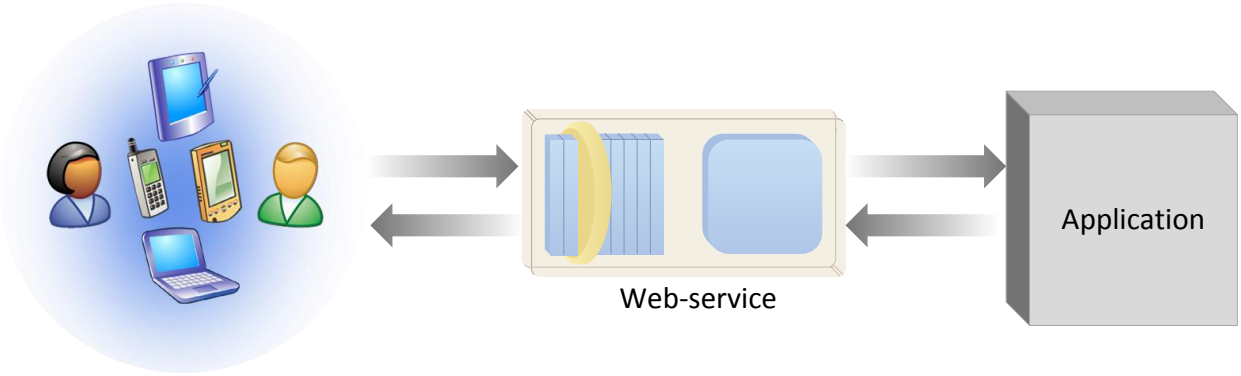



Рис. 50. Схема зв’язку між клієнтським додатком і кодом програми

На теперішній час створено багато web-систем, організованих на базі простої клієнт-серверної архітектури. У цьому випадку основою сайту є процес, який має доступ до локальної файлової системи зберігання документів за допомогою Uniform Resource Locator (URL), котрий визначає місце знаходження документа. На рис. 51 наведена схема обробки запита клієнту зі стандартним зв'язком між браузером та web-сервером з використанням протоколу Hypertext Transfer Protocol (HTTP).

Поряд з цим існує швидко зростаюча група web-систем, які пропонують загальні послуги віддаленого додатку без негайної взаємодії з кінцевими

користувачами. Ця організація веде до концепції web-сервісів.

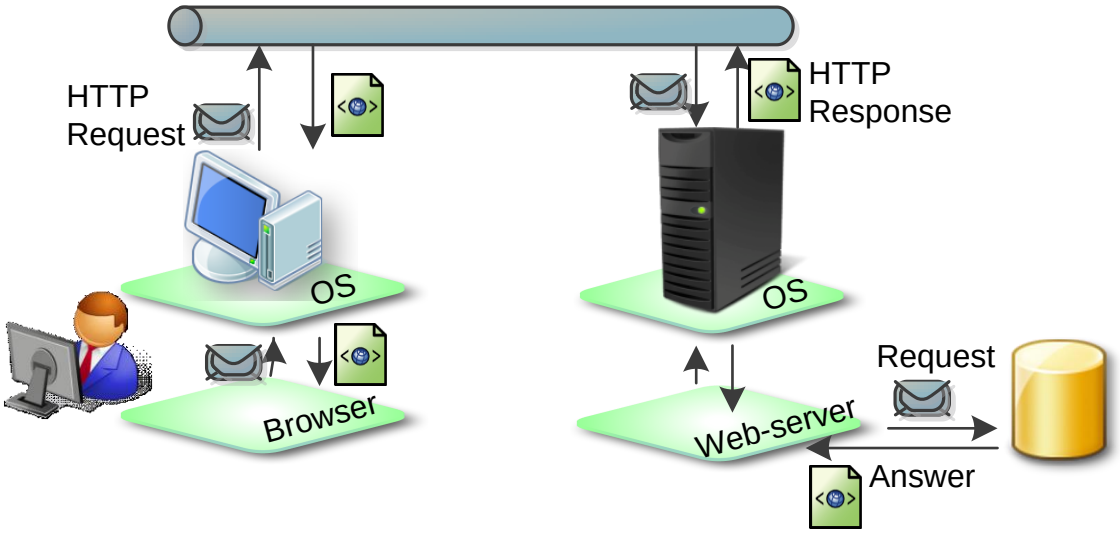



Рис. 51. Схема зв'язку між браузером та web-сервером з використанням HTTP [Tidwell, 2001]

Сервіси рівню додатка – механізми публікації, управління, пошуку та вибірки контенту, доступ до яких здійснюється на базі протоколу HTTP та HTML, клієнтські додатки – web-браузери (web browsers), які розуміють ці стандарти і можуть зв’язуватися з сервісами (рис.52). Такий зв'язок не залежить від платформ, на базі яких функціонують додатки, бо взаємодія здійснюється на рівні протоколів.

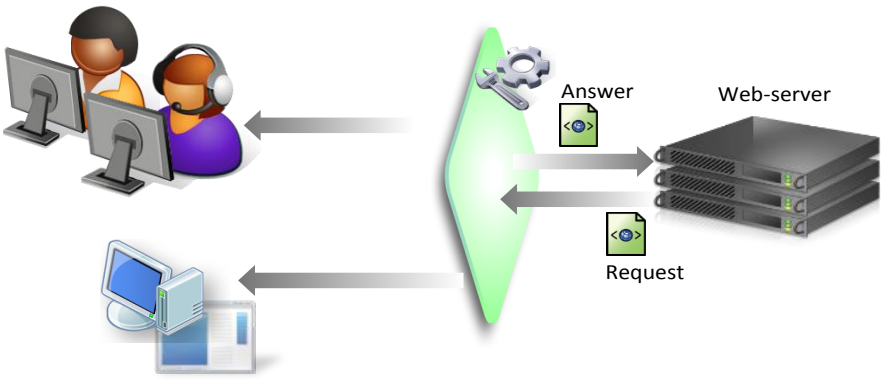



Рис.52. Концепція застосування web-сервісів

Одним з перших підходів було введення механізму CGI (Common Gateway Interface ), який визначає спосіб виконання web-сервером програми, яка отримує дані користувача в якості вхідних параметрів (рис. 53). Призна-

чені дані приходять з HTML-форми. Після завершення заповнення форми назва програми і зібрані значення параметрів передаються на сервер.

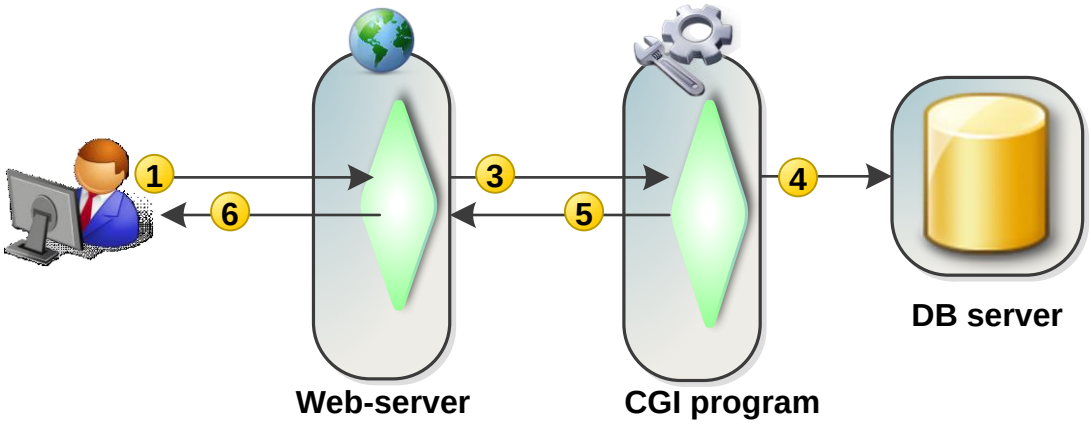



Рис. 53. Принцип взаємодії додатків за допомогою CGI

Після обробки документа сервер передає його на сторону клієнта.

Приклад запиту програмі CGI [CGI]

```
<html><head><title>Guestbook Form</title></head>
<body>
<form action="post.cgi" method="POST">
Your Name: <input type="text" name="name"><br>
Email Address: <input type="text" name="email"><br>
Comments:<br>
<textarea name="comments" rows="5"
  cols="60"></textarea><br>
<input type="submit" value="Send">
</form>
</body>
</html>
```

CGI - скрипт

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
```

```
use strict;
print header;
print start_html("Thank You");
print h2("Thank You");
my %form;
foreach my $p (param()) {
        $form{$p} = param($p);
        print "$p = $form{$p}<br>\n";
}
print end_html;
```

На сервері, де знаходиться сайт, повинно бути дозволено виконання CGI –скриптів, і проблема такого підходу складається в тому, що скрипт, як і будь-яка інша програма, може виконувати системні команди на сервері, що становить потенційну загрозу безпеці [CGI], [CGI2].

Наступним важливим удосконаленням є те, що сервери стали спроможні обробляти документ до передачі його клієнту. Документ може містити серверний скрипт, який виконується на сервері. Результат виконання сценарію відправляється разом з іншою частиною документа до клієнта. Сам скрипт не передається. Серверні сценарії формують документ, додаючи результати виконання сценарію.

Серверна обробка web-документів все частіше потребує більшої гнучкості. Багато web-сайтів організовані на трьохрівневій архітектурі, яка складається з web-сервера, сервера додатків і баз даних. Наприклад, сервер може приймати запити клієнтів, робити пошук в базі даних щодо відповідних продуктів, а потім побудувати інтерактивну web-сторінку із списком знайдених продуктів.

У відповідності з принципами СОА до Web-сервісу пред’являються наступні вимоги: доступність через мережу Internet або intranet; використання стандартизованої системи повідомлень на базі XML; незалежність від ОС або

мови програмування; опис інтерфейсу-контракту на базі XML; простий механізм публікацій та пошуку.

Web-сервіс повинен містити в собі всю інформацію про себе, тобто бути самодостатньо–описуваним. Якщо публікується новий web-сервіс, потрібна публікація і інтерфейсу до цього сервісу. Наприклад, при створенні сервісу використовується протокол SOAP, до нього додається інтерфейс написаний на базі XML, який визначає всі відкриті методи, аргументи, результати.

Web-служба повинна бути доступною для пошуку. У разі створення сервісу повинен бути доступним простий механізм, що забезпечує знаходження цього сервісу зацікавленими сторонами (сервісами або додатками). Такий механізм може бути створений на базі централізованого або децентралізованого реєстру.

Типова схема роботи системи виглядає наступним чином (рис. 54).

1. Додаток ініціює пошук необхідного сервісу (наприклад, сервісу певної компанії) в централізованій директорії web-сервісів. Директорія повертає інформацію щодо сервісу, яка включає вказівник на місце, де розташований опис сервісу (інтерфейс).

2. Додаток зв'язується з сервісом компанії та отримує його опис.

3. Керуючись описом додаток створює запит необхідної функції.

На теперішній час сучасна технологія web-сервісів дозволяє автоматизувати цей процес частково: отримання набора сервісів з централізованого реєстру та роботу з сервісом на базі його опису. Автоматизація вибору сервісу лишається за людиною.

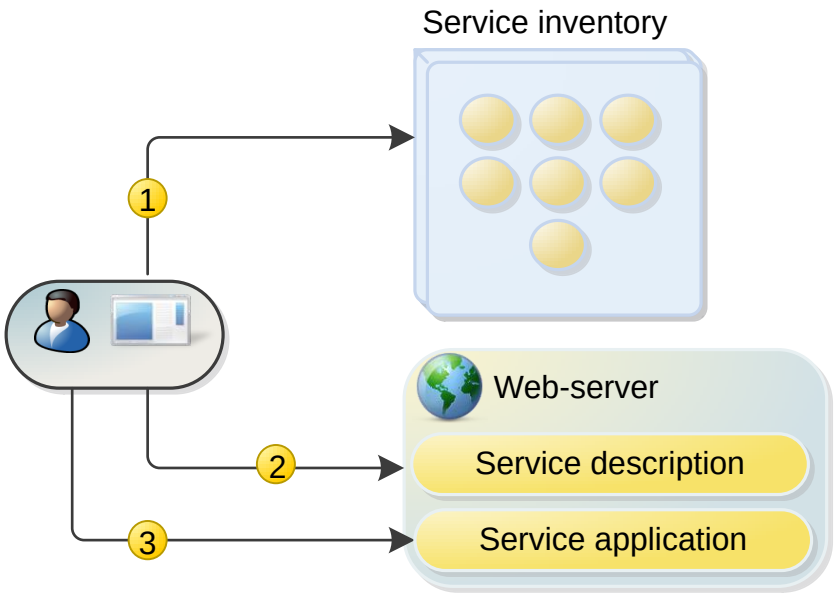



Рис. 54. Схема роботи системи автоматизації інтеграції додатків

Окрім цього, лишається проблема автоматизації бізнес зв'язків (business relationships).

Незважаючи на ряд особливостей переважна більшість систем підтримує загальний набір технологій: SOAP, HTTP, WSDL, UDDI (табл. 16).

Архітектура web-сервісів розглядається з точок зору ролей та стеку протоколів. З точки зору ролей відрізняють три основні архітектури:

- *Service provider* (провайдер) - забезпечує виконання сервісу, доступ через Інтернет;

- *Service requestor* (споживач) - використовує сервіс шляхом з'єднання та передачі XML-запиту;

- *Service registry* (централізована директорія (реєстр)) - дозволяє розробникам публікацію нових та пошук існуючих сервісів.

Основні протоколи web-сервісів

Таблиця 16

| Технологія | Призначення |
|---|---|
| WSDL | Заснований на XML формат опису web-служб, її методів, типів даних параметрів і значення, що повертається, а також підтримуваних методів комунікації. |

| | |
|---|---|
| HTTP, SMTP | Комунікаційні протоколи, які служать для відправки запитів w?-eb-службі через Інтернет. |
| SOAP, XML-RPC, XML | Заснований на XML формат кодування інформації в запиті, відправленому web-службі. Наприклад, SOAP визначає способи представлення величин різних типів даних. |
| UDDI | UDDI (Universal Description, Discovery, and Integration) протокол є центральним елементом групи відповідних стандартів, які включають web-служби стеку. Специфікація UDDI визначає стандартний метод для публікації та виявлення мережевих програмних компонентів сервіс-орієнтованої архітектури. UDDI описує реєстр web-служб і програмних інтерфейсів для публікації, пошуку і керування інформацією про послуги, описані в ньому. Офіційна специфікація UDDI формально описує web-служби, структури даних і поведінку реєстру, що відповідає стандарту UDDI. |

Стек протоколів містить чотири основних шари.

*Service transport* (передача***)*** - відповідає за передачу повідомлень між додатками та включає протоколи HTTP, Simple Mail Transfer Protocol (SMTP), file transfer protocol (FTP), Blocks Extensible Exchange Protocol (BEEP).

*XML messaging* (повідомлення) − відповідає за формування та розбирання повідомлень на базі XML, включає протоколи XML-RPC та SOAP.

*Service description* (опис) − відповідає за інтерфейс доступу до служби, опис сервісу здійснюється на базі Web Service Description Language (WSDL).

*Service discovery* (пошук) − відповідає за централізацію сервісів в загальному реєстрі, забезпечення механізмів публікації та пошуку. Відкриття сервісів здійснюється на базі протоколу Universal Description, Discovery, and Integration (UDDI).

### 8.2.1. Протокол XML-RPC

Перед розглядом протоколів подання повідомлень, які базуються на стандарті XML, варто приділити увагу особливостям даного протоколу.

Розширювана мова розмітки XML (eXtensible Markup Language) належить до мета-мов, тобто вона використовується для опису інших мов розмітки, що є підмножинами XML. Таким чином, у кожній підмножині можуть бути визначені свої власні теги. При цьому велика ймовірність того, що різні мови розмітки будуть використовувати однакові назви тегів, які наповнені зовсім різним змістом. Для того щоб уникнути подібних конфліктів і використовується поняття простору імен, що однозначно вказує, до якого XML діалекту належить даний документ.

Схема визначення значення кожного з тегів у документі XML

```
<?xml version=»1.0»?>
<xs:schema  xmlns:xs=http://www.w3.org/2001/XMLSchema
xmlns=“document" >
<xs:element name = “DOCUMENT”>
<xs:element name=“CUSTOMER"> </xs:element>
</xs:element>
</xs:schema>
<?xml version=“1.0”?>
<DOCUMENT
xmlns=“document”
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
Xsi:schemaLocation=“order.xsd”>
<DOCUMENT>
    <CUSTOMER>sam smith</CUSTOMER>
    <CUSTOMER>sam smith</CUSTOMER>
</DOCUMENT>
```

З використанням абстрактної мета-мови XML побудована велика кількість стандартів у різноманітних галузях. Звичайно, при описуванні якого-небудь документу залучається деякий словник, котрий потім може бути описаний з використанням XSD. Стандарт в даному розумінні зводиться до поняття словника термінів, які складають деякий кінцевий набір тегів та їх ат-

рибутів. Одним з перших таких словників являється мова XHTML, яка підтримується більшою кількістю браузерів. Існують комерційні словники, такі як CommerceML, які використовуються для передачі даних, орієнтованих на торговельну діяльність, ці словники містять у собі опис системи замовлень, постачальників, продуктів та інше. Були створені й більш спеціалізовані словники, наприклад, протокол передачі даних SOAP, мова описання контрактів веб-служб WSDL.

Для протоколів передачі повідомлень відзначимо два основних стандарти *XML-RPC* і *SOAP*, які використовують словники, побудовані з застосуванням стандарту XML.

*XML-RPC* - це простий протокол, що використовує XML повідомлення для виконання RPC. Запити кодуються в XML і відправляється з використанням транспортного протоколу HTTP POST. При цьому XML повідомлення-відповіді містяться в тілі відповіді HTTP. Оскільки XML-RPC протокол не залежить від платформи, це дозволяє різним додаткам взаємодіяти. Наприклад, Java-клієнт використовуючи XML-RPC може взаємодіяти зі сервером Perl.

*XML-RPC.NET* реалізує XML-RPC служби, які можуть працювати у web-сервері Microsoft IIS. XML-RPC Service будується з використанням будь-якої з CLS-сумісних мов (наприклад, C #, VB.Net, і т.д.) шляхом створення класу, який походить від класу XmlRpcService з використанням бази та методів, що відображаються на XML-RPC з атрибутом XmlRpcMethod.

Типовий приклад запиту на базі протоколу XML-RPC

```
<?xml version="1.0"?>
<methodCall>
 <methodName>examples.getStateName</methodName>
 <params>
  <param>
  <value><i4>40</i4></value>
  </param>
```

```
 </params>
</methodCall>
```

Відповідь на  наданий запит

```
<?xml version="1.0"?>
<methodResponse>
 <params>
  <param>
  <value><string>South Dakota</string></value>
  </param>
 </params>
 </methodResponse>
```

Формат помилки XML-RPC

```
 <?xml version="1.0"?>
<methodResponse>
 <fault>
  <value>
  <struct>
  <member>
  <name>faultCode</name>
  <value><int>4</int></value>
  </member>
  <member>
  <name>faultString</name>
  <value><string>Too many parameters.</string></value>
  </member>
  </struct>
  </value>
  </fault>
```

</methodResponse>

Можливими альтернативами можуть бути: передача документу нестандартного формату засобами HTTP GET/POST, або використання протоколу SOAP (Simple object access protocol). [XML-RPC]

### 8.2.2. Протокол SOAP

*SOAP* (Simple Object Access Protocol - простий протокол доступу до об'єктів) являється основним протоколом презентації повідомлень для вибору web-служб. Основна ідея стандарту SOAP полягає у тому, що повідомлення повинні бути закодовані в стандартизованому XML-форматі. Можна сказати, що формат SOAP ідеально підходить для технології RPC (Remote Procedure Call - виклик віддаленої процедури), оскільки SOAP-повідомлення містять параметри, які направляються клієнтом, або відповідною службою, що повертаються.

Приведений нижче фрагмент коду містить просте SOAP-повідомлення - повідомлення-запит до web-служби GetIPForHostname. Його основна частина поміщена в спеціальний тег <soap:Envelope>. В даному випадку відсилається один параметр (hostName), значенням якого є рядок www.aaa.com.

```
<?xml version="1.0" encoding-"utf-8"?>
<soap:Envelope  mlns:xsi=http://www.w3.org/2001/XMLSchema-
instance
xmlns:xsd=http://www.w3.org/2001/XMLSchema
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<GetIPForHostname xmlns-"http://www.bostontechni>.com/webservices?">
<hostName>www.aaa.com</hostName>
</GetIPForHostname>
</soap:Body>
</soap:Envelope>
```

Повідомлення SOAP складається з двох частин - заголовка та тіла,

складаючи структуру, яка має назву "конверт SOAP"(SOAP envelope). SOAP конверт не містить адресу одержувача, тому що доставка повідомлення є задачею іншого протоколу. Наприклад, коли повідомлення SOAP пов'язані з HTTP, одержувач буде вказаний у вигляді URL.

SOAP-конверт

SOAP-заголовок

Элемент заголовка 1

Элемент заголовка 2

…

Элемент заголовка N

Тіло SOAP

Элемент тіла N

…

Элемент тіла 2

Элемент тіла 1

Приклад SOAP-запита на сервер інтернет-магазину

```
<?xml version='1.0' encoding='UTF-8'?>
<soap:Envelope
xmlns:soap=http://www.w3.org/2001/09/soap-envelope/
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
  <getProductDetails xmlns="http://warehouse.example.com/ws">
  <productID xsi:type="xsd:string">12345</productID>
  </getProductDetails>
  </soap:Body>
 </soap:Envelope>
```

Приклад відповіді

```
<?xml version='1.0' encoding='UTF-8'?>
<soap:Envelope xmlns:soap=http://www.w3.org/2001/09/soap-envelope/
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd=http://www.w3.org/2001/XMLSchema>
 <soap:Body>
 <getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
 <getProductDetailsResult>
 <productID xsi:type="xsd:int">12345</productID>
  <productName xsi:type="xsd:string">Стакан граненый</productName>
 <description xsi:type="xsd:string">Стакан граненый. 200
мл.</description>
   <price xsi:type="xsd:int">9.95</price>
   <inStock xsi:type="xsd:bool">true</inStock>
   </getProductDetailsResult>
   </getProductDetailsResponse>
   </soap:Body>
   </soap:Envelope>
```

До недоліків використання SOAP слід віднести те, що для передачі повідомлень він збільшує їх розмір та понижує швидкість обробки. Альтернатива в цьому випадку – пряма пересилка документів XML використовуючи HTTP.

Важливо відзначити, що XML - повідомлення можуть використовуватися не тільки при передачі по протоколу HTTP, але також при пересиланні за допомогою сокетів, іменованих каналів і навіть по протоколу SMTP електронної пошти.

```
POST /demo/MSDN/PerfCounter.asmx HTTP/1.1
Connection: Keep-Alive
Content-Length: 150
Content-Type: text/xml
```

```
Host: localhost
User-Agent: MS Web Services Client Protocol 1.0.2204.19
SOAPAction: "http://tempuri.org/PerfCounters"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/1999/XMLSchema">
 <soap:Body>
 <PerfCounters xmlns="http://tempuri.org/"/>
 </soap:Body>
</soap:Envelope>
```

Окрім повідомлень SOAP, для обміну даними з web-службами .NET можна використовувати методи GET і POST протоколу HTTP. Теоретично при передачі інформації методом POST ви можете, як і раніше, застосовувати формат SOAP, але в цьому випадку дані простіше передавати у вигляді набору ім'я-значення без вказівки їх типу.

### 8.2.3. Мова WSDL для визначення web-сервісів і доступу до них

Кожна web-служба надає документ WSDL (Web Service Description Language - мова опису web-служби), в якому описуються всі необхідні клієнту відомості про цю службу. WSDL-документ служить тим же цілям, що і файл IDL (Interface Definition Language - мова визначення інтерфейсу) для компоненту CORBA або COM - він визначає інтерфейс web-служби. Вказаний документ, по суті, є контрактом між клієнтом і web-службою, де декларується, що «якщо ви викличете такий-то метод з такими-то параметрами, то як величина, що повертається, одержите такі-то дані».

У багатьох відношеннях web-служби навіть простіші, ніж створювані

для CORBA або COM компоненти. Наприклад, в web-службах відсутня можливість підтримки декількох інтерфейсів - кожен клас web-служби забезпечує тільки один набір відкритих (public) методів. З другого боку, документ WSDL дещо складніший за свій IDL-еквівалент, оскільки він є платформонезалежним і підтримує комунікаційні протоколи, відмінні від SOAP і HTTP. Це означає, що кожен WSDL-файл для web-служби .NET містить значний об'єм стереотипного коду, що служить для забезпечення підтримки базового рівня комунікації (відповідно до протоколу SOAP або методів GET і POST протоколу HTTP).

WSDL-документ має заснований на XML формат, відповідно до якого інформація підрозділяється на п'ять груп. Перші три групи є абстрактними визначеннями, незалежними від особливостей платформи, мережі або мови, а дві останні групи включають конкретні описи.

У документі WSDL визначається web-сервіс за допомогою наступних основних елементів (табл.17).

Основні елементи мови WSDL

Таблиця 17

| Елемент | Визначає |
|---|---|
| <portType> | Методи, що надаються web-сервісом |
| <message> | Повідомлення, що використовуються web-сервісом |
| <types> | Типи даних, що використовуються web-сервісом |

Основна структура документа WSDL виглядає наступним чином:

```
<definitions>
<types>
 визначення типів.....
</types>
<message>
 визначення повідомлення....
</message>
<portType>
```

визначення порту.....
</portType>
<binding>
визначення зв’язків .....
</binding>
</definitions>

Елемент <portType> є найбільш важливим елементом у WSDL. Він визначає сам web-сервіс, що надається їм операції і використовані повідомлення. Елемент <portType> можна порівняти з бібліотекою функцій (модулем, класом) у традиційній мові програмування.

Кожне WSDL повідомлення може містити одну або декілька частин. Ці частини можна порівняти з параметрами виклику функцій у традиційній мові програмування.

Елемент <types> визначає тип даних, що використовуються web-сервісом. Для максимальної платформо-незалежності WSDL використовує синтаксис XML Schema для визначення типів даних.

Елемент зв’язку <binding> визначає формат повідомлення і деталі протоколу для кожного порту.

Наведемо фрагмент WSDL-документу

```
<message name="getTermRequest">
 <part name="term" type="xs:string"/>
</message>
<message name="getTermResponse">
<part name="value" type="xs:string"/>
</message>
<portType name="glossaryTerms">
 <operation name="getTerm">
  <input message="getTermRequest"/>
  <output message="getTermResponse"/>
  </operation>
  </portType>
```

У цьому прикладі елемент portType визначає "glossaryTerms" як ім'я порту, і "getTerm" як ім'я операції. Операція "getTerm" має вхідне повідомлення, зване "getTermRequest", і вихідне повідомлення - "getTermResponse".

Елементи message визначають частини кожного повідомлення та асоційовані типи даних.

Якщо порівнювати з традиційним програмуванням, то glossaryTerms - бібліотека функцій, а "getTerm" - функція з параметром "getTermRequest", яка повертає getTermResponse після виконання.

Механізми зв'язування на сервісі показані на рис.55, загальна структура WSDL 1.1 і 2.0 файлів – на рис.56.

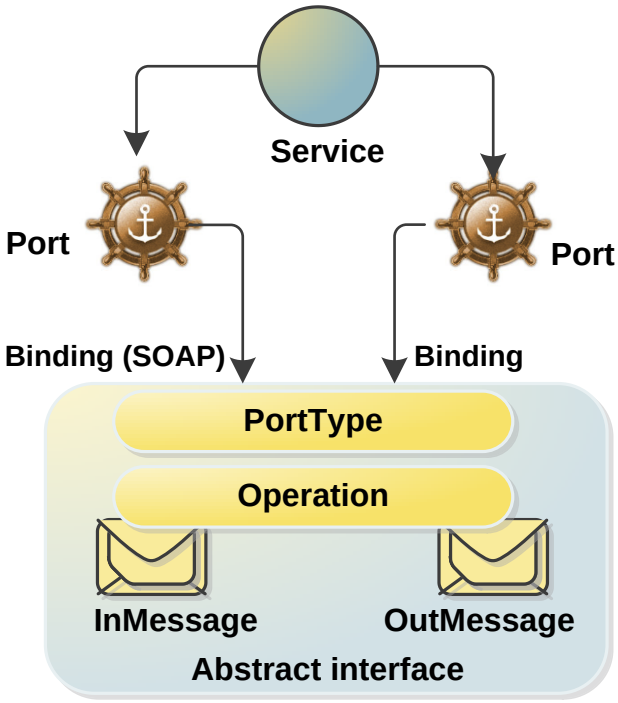



Рис. 55. Механізми зв'язування на сервісі

Як розширення IDL, WSDL надає інструменти для створення сумісного клієнта і сервера за алфавітом. Забезпечує підтримку інструментів для розробки додатків зверху вниз, знизу вгору і "на зустріч у середині". Дозволяє визначити стандартизовані інтерфейси служб у галузі. Дозволяє сформувати опис сервісу у стандартній формі, що забезпечує його динамічне знаходження та зв'язок з іншими сервісами.

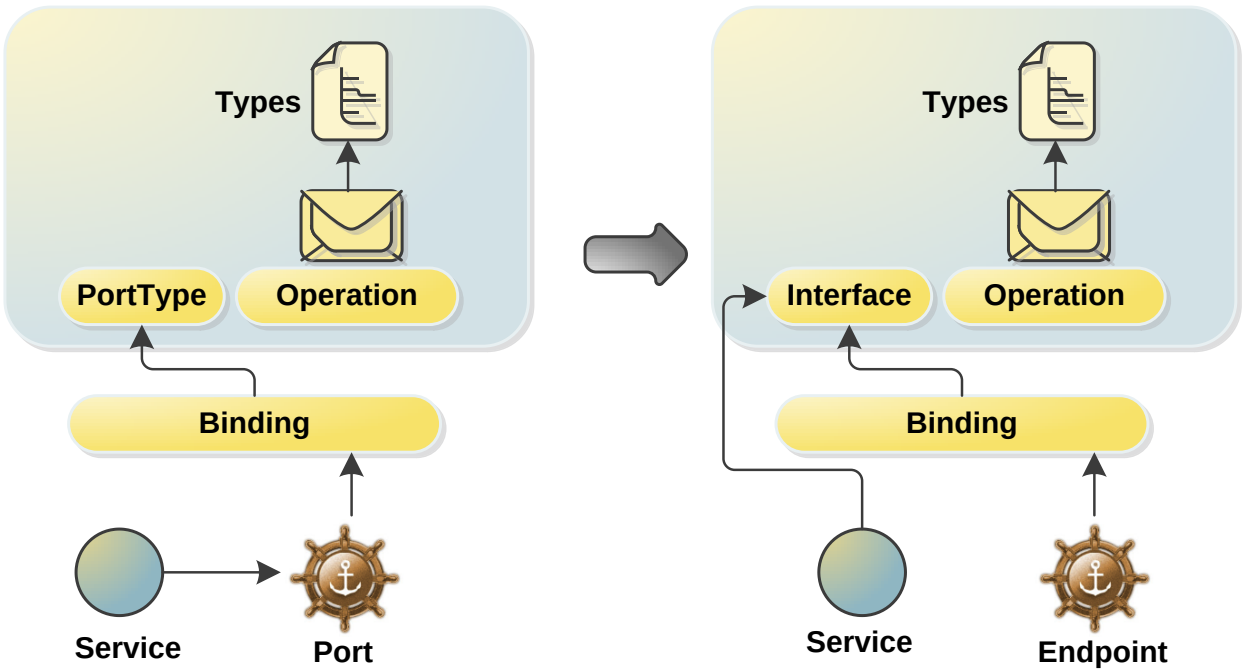



Рис. 56. Загальна структура WSDL 1.1 і 2.0 файлів

WSDL файли описують послуги знизу вгору. Тобто вони починаються з використовуваних типів даних і закінчуються місцезнаходженням (адреси) служби. Крім того вони мають три шари опису.

Перший шар описує інтерфейс сервісу. Цей інтерфейс ("порт типу" в WSDL 1.1) може складатися з однієї або кількох операцій з вхідними і вихідними параметрами, що використовують типи, зазначені в першому розділі файлу WSDL1.1. В WSDL службі параметри були визначені в <message> секції, а в WSDL 2.0 вони визначаються так саме, як тип в <types> розділі.

Другий шар є "обов'язковим" для web-служби, він визначає протокол і формат, для якого web-служби операцій надаються.

Третій шар визначає фізичне місце розташування (адреса, URL), де web-сервіс є доступним.

Розглянемо простий приклад файлу WSDL 1.1. Цей файл WSDL визначає сервіс під назвою CustomerService, який надає одну операцію під назвою getCustomerAddress (). Його вхідний параметр- ідентифікатор клієнта типу long, на виході формується структура з трьох рядків атрибутів: вулиця, місто і поштовий індекс.

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <definitions name="CustomerService"
3 targetNamespace="http://soa-in-practice.com/wsdl"
4 xmlns:tns="http://soa-in-practice.com/wsdl"
5 xmlns:xsd1="http://soa-in-practice.com/xsd"
6 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
7 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
8 xmlns="http://schemas.xmlsoap.org/wsdl/">
9
10 <types>
11 <xsd:schema
12 targetNamespace="http://soa-in-practice.com/xsd"
13 xmlns="http://soa-in-practice.com/xsd">
14
15 <xsd:element name="getCustomerAddress">
16 <xsd:complexType>
17 <xsd:sequence>
18 <xsd:element name="customerID" type="xsd:long"/>
19 </xsd:sequence>
20 </xsd:complexType>
21 </xsd:element>
22
23 <xsd:element name="getCustomerAddressResponse" type="Address"/>
24 <xsd:complexType name="Address">
25 <xsd:sequence>
26 <xsd:element name="street" type="xsd:string"/>
27 <xsd:element name="city" type="xsd:string"/>
28 <xsd:element name="zipCode" type="xsd:string"/>
29 </xsd:sequence>
30 </xsd:complexType>
```

```
31
32 </xsd:schema>
33 </types>
34
35 <message name="getCustomerAddressInput">
36 <part name="params" element="xsd1:getCustomerAddress"/>
37 </message>
38 <message name="getCustomerAddressOutput">
39 <part name="params" element="xsd1:getCustomerAddressResponse"/>
40 </message>
41

42 <portType name="CustomerInterface" >
43 <operation name="getCustomerAddress">
44 <input message="tns:getCustomerAddressInput" />
45 <output message="tns:getCustomerAddressOutput" />
46 </operation>
47 </portType>
48
49 <binding name="CustomerSOAPBinding"
50 type="tns:CustomerInterface" >
51 <soap:binding style="document"
52 transport="http://schemas.xmlsoap.org/soap/http" />
53 <operation name="getCustomerAddress">
54 <soap:operation
55 soapAction="http://soa-in-practice.com/getCustomerAddress" />
56 <input>
57 <soap:body use="literal" />
58 </input>
59 <output>
```

```
60 <soap:body use="literal" />
61 </output>
62 </operation>
63 </binding>
64
65 <service name="CustomerService" >
66 <port name="CustomerPort"
67 binding="tns:CustomerSOAPBinding">
68 <soap:address
69 location="http://soa-in-practice.com/customer11"/>
70 </port>
71 </service>
72
73 </definitions>
```

Розділ <service> визначає сервіс з ім'ям CustomerService, доступний за адресою, яка надається з обов'язковим ім’ям CustomerSOAPBinding. Префікс "tns:" - простір імен.

Розділ <binding> включає протокол та описання формату повідомлення, які використовуються для надання послуг. Так CustomerSOAPBinding починається з визначення того, який контракт використовується для взаємодії з сервисом (CustomerInterface).

Розділ <portType> визначає контракт-інтерфейс CustomerInterface. В даному випадку він визначає одну операціюї getCustomerAddress. Ця служба надсилає запит і отримує відповідь, що визначається шляхом зазначення вхідного повідомлення (getCustomerAddressInput) і вихідного повідомлення (getCustomerAddressOutput). Інші повідомлення можуть бути повідомленнями про помилки.

Розділи <message> визначають окремі повідомлення, використовуючи ідентифікатори, згадані в розділі <portType>. Обидві процедури getCustomer AddressInput і getCustomerAddressOutput використовують тип даних визначе-

ний у <types> розділі.

Розділ <types> визначає використовувані типи даних: в цьому випадку використовується вхідний параметр CustomerID типу long і вихідний параметр адреси, який представляє собою структуру, і запис з трьох атрибутів рядка. Всі види мають свої власні імена, xsd1.

Відповідний файл WSDL для версії 2.0 (з використанням SOAP 1.2):

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <description
3 targetNamespace="http://soa-in-practice.com/wsdl"
4 xmlns:tns="http://soa-in-practice.com/wsdl"
5 xmlns:xsd1="http://soa-in-practice.com/xsd"
6 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
7 xmlns:wsoap="http://www.w3.org/2006/01/wsdl/soap"
8 xmlns:wsdlx="http://www.w3.org/2006/01/wsdl-extensions"
9 xmlns="http://www.w3.org/2006/01/wsdl">
10
11 <types>
12 <xsd:schema
13 targetNamespace="http://soa-in-practice.com/xsd"
14 xmlns="http://soa-in-practice.com/xsd">
15
16 <xsd:element name="getCustomerAddress">
17 <xsd:complexType>
18 <xsd:sequence>
19 <xsd:element name="customerID" type="xsd:long"/>
20 </xsd:sequence>
21 </xsd:complexType>
22 </xsd:element>
23
24 <xsd:element name="getCustomerAddressResponse" type="Address"/>
```

```
25 <xsd:complexType name="Address">
26 <xsd:sequence>
27 <xsd:element name="street" type="xsd:string"/>
28 <xsd:element name="city" type="xsd:string"/>
29 <xsd:element name="zipCode" type="xsd:string"/>
30 </xsd:sequence>
31 </xsd:complexType>
32
33 </xsd:schema>
34 </types>
35

36 <interface name = "CustomerInterface" >
37
38 <operation name="getCustomerAddress"
39 pattern="http://www.w3.org/2006/01/wsdl/in-out"
40 style="http://www.w3.org/2006/01/wsdl/style/iri"
41 wsdlx:safe = "true">
42 <input messageLabel="In"
43 element="xsd1:getCustomerAddress" />
44 <output messageLabel="Out"
45 element="xsd1:getCustomerAddressResponse" />
46 </operation>
47
48 </interface>
49
50 <binding name="CustomerSOAPBinding"
51 interface="tns:CustomerInterface"
52 type="http://www.w3.org/2006/01/wsdl/soap"
53 wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP">
```

```
54
55 <operation ref="tns:getCustomerAddress"
56 wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>
57
58 </binding>
59
60 <service name="CustomerService"
61 interface="tns:CustomerInterface">
62
63 <endpoint name="CustomerEndpoint"
64 binding="tns:CustomerSOAPBinding"
65 address="http://soa-in-practice.com/customer20"/>
66
67 </service>
68
69 </description>
```

Основні відмінності

- Кореневий елемент XML називається <description> замість <definiti>-ns>.
- Розділ <service> використовує термін "кінцева точка" замість "порт".
- Всередині розділу <binding> лінії 52, 53 і 56 визначають конкретний протокол (SOAP 1.2, заснований на HTTP), використовуючи HTTP GET.
- Розділ <portType> замінено розділом <interface>. Він використовує більш конкретні моделі обміну повідомленнями, які визначають низький рівень впорядкування повідомлень.
- Розділ <message> більше не доступний. Замість цього операції безпосередньо відносяться до типів даних (визначених у розділі <types>). Повідомлення, які відправляються, визначаються з MEP і типів, зазна-

чених у розділі <interface>.

- Різні простори імен використовуються для визначення WSDL і SOAP. Перший шар (тип повідомлення та portTypes для WSDL 1.1 і типи інтерфейсів WSDL 2.0) є синтаксичний інтерфейс web-служби, із зазначенням підпису служби. Інші шари визначають технічні деталі.

### 8.2.4. Сервіс UDDI для реєстрації та пошуку web-сервісів

UDDI (Universal Description Discovery & Integration) − це з одного боку стандарт, спрямований на інтеграцію web-служб, з другого − відкрите середовище для реєстрації та розміщення бізнес-сервісів з забезпеченням механізму динамічного пошуку. Відповідно до звичайних веб-сайтів, процедура пошуку може бути здійснена двома способами: безпосереднього завдання адреси сервісу та використання механізму пошуку (search engine). Ядром являється бізнес реєстрація − xml-файл, який використовується для опису бізнес об'єкту і його служб. Таким чином UDDI можна розглядати як службу DNS для веб-додатків.

В цілому ідея складається в побудові загального каталогу, директорії, до якої можливе здійснення доступу інших сервісів з використанням набору xml-стандартів. Підтримуються три типа рівня деталізації опису сервісів, які відповідно називаються білими, жовтими та зеленими сторінками. Білі сторінки надають можливість пошуку сервісу тільки за ім’ям. Жовті − за категорією, заданою кодом системи класифікації (напр. NAICS - North-American Industrial Classification System, UN Standard Products and Services Codes, Geolocation), зелені − описують деталі доступу до веб-сервісу з інформацією щодо його зв’язування з протоколами та точками доступу. Якщо сервіс має декілька зв’язувань (bindings), які визначаються в його WSDL-описі, сервіс повинен мати декілька зелених сторінок.

UDDI визначає функціонування служби реєстру, яка включає: структури даних для реєстрації підприємств, технічних характеристик, сервісів та

обслуговування кінцевих точок на базі технічних характеристик; SOAP API; правила функціонування глобального реєстру.

Коли до серверу надходить запит, запускається програма, вказана в запиті з параметрами. Тоді програма просто виконує свою роботу і, як правило, повертає результати у вигляді документу, який відправляється назад в браузер користувача і відображається.

Проблеми, які вирішує специфікація UDDI:

- можливість виявити підприємця вже існуючого в мережі;
- визначення способів співпраці з знайденим підприємцем;
- пошук нових замовників і спрощення доступу до вже існуючих ;
- розширення пропозицій і збільшення ринкової досяжності;
- рішення проблеми виникаючих бар'єрів при участі в глобальній економіці Internet;
- опис сервісів і ділових процесів програмно в єдиному, відкритому і перспективному середовищі.

UDDI використовується при створенні різних платформ і програмних продуктів, таких фірм як Dell, Fujitsu, HP, Hitachi, IBM, Intel, Microsoft, Oracle, SAP і Sun.

UDDI XSD визначають декілька видів основної інформації, які необхідно знати користувачам і додаткам для використання певної веб-служби. Разом вони утворюють базу метаінформації в рамках реєстрів UDDI (рис.57), яка містить: опис служби бізнес-функції (*BusinessService*); інформацію про організацію, яка публікує колекцію бізнес-служб (*BusinessEntity*); технічну інформацію, яка необхідна для використання бізнес-служби (*Bindi-?gTemplate*); технічну модель, яка застосовується для репрезентації інформації, що повторно використовується (напр.: тип сервісу, протокол, система категоризації) (TModel).

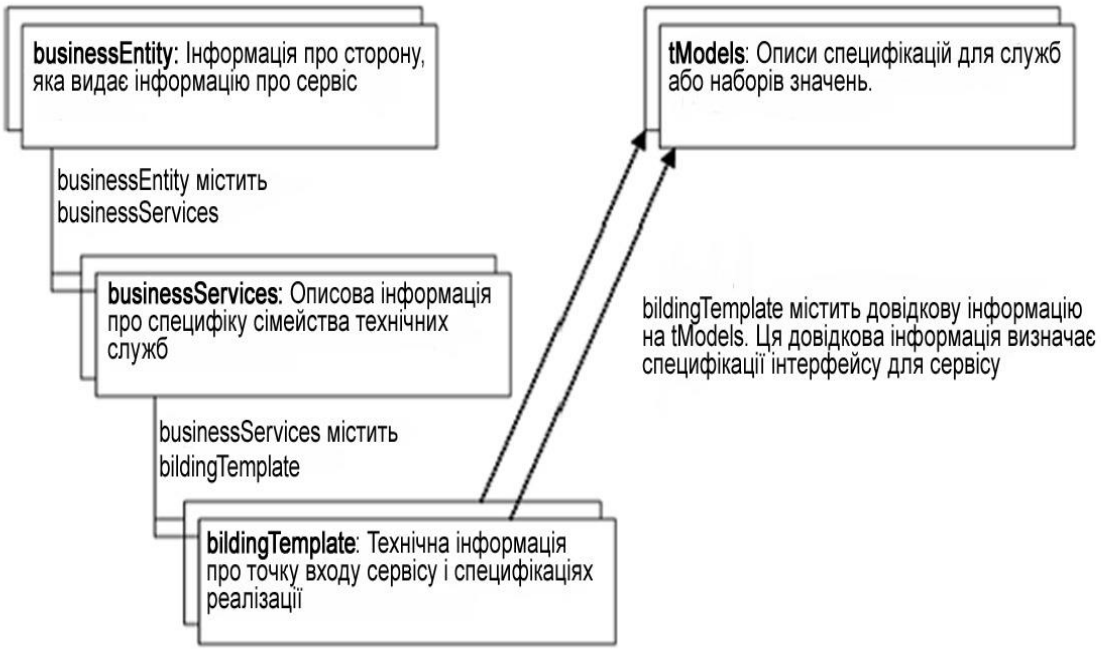



Рис.57. Модель метаінформації в рамках реєстрів UDDI

У межах системного реєстру UDDI кожній структурі даних призначають унікальний ідентифікатор згідно зі стандартною схемою.

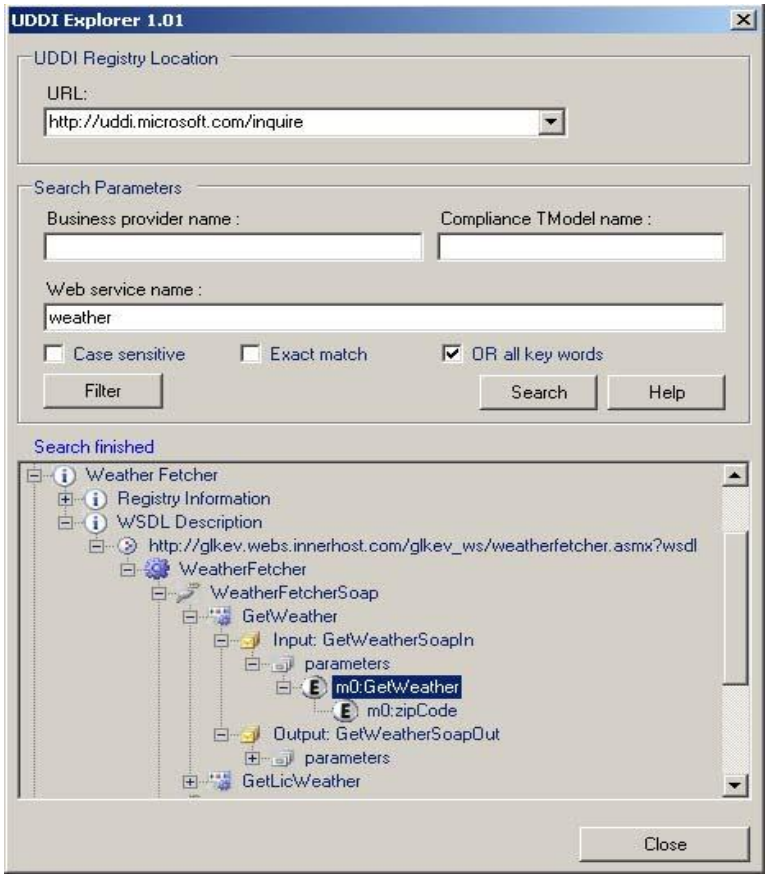



Рис.58. Вікно пошуку доступних сервісів

### 8.3. REST підхід до побудови сервісно-орієнтованих додатків

REST (Representational State Transfer) − це архітектурний стиль, який будується на існуючих, добре відомих W3C-стандартах, таких, як HTTP, URI (Uniform Resource Identifier), XML і RDF (Resource Description Format). У REST-сервісах акцент зроблено на доступ до ресурсів, а не на виконання віддалених сервісів,- в цьому їх кардинальна відмінність від SOAP-сервісів. Якщо SOAP-клієнти звертаються з запитом на виконання процедури на сервері, то REST-клієнти просто вимагають сам ресурс. Наприклад, замість того, щоб запитувати віддалене виконання функції для знаходження потрібного формуляра замовлення, проводиться запит цього формуляру, приблизно так само, як Web-сторінки. Дані повинні передаватися у вигляді невеликої кількості стандартних форматів (наприклад, HTML, XML, JSON). [REST].

JSON [REST2] (Java Script Object Notation) — це легкий формат обміну даними між комп'ютерами. JSON базується на тексті, і може бути з легкістю прочитаним людиною. Формат дозволяє описувати об'єкти та інші структури даних. JSON головним чином використовується для передачі структурованої інформації через мережу (завдяки процесу, що називають серіалізацією). JSON знайшов своє головне призначення у написанні веб-додатків, а саме при використанні технології AJAX. JSON виступає як заміна формату XML під час асинхронної передачі структурованої інформації між клієнтом та сервером. При цьому перевагою JSON перед XML являється те, що він дозволяє зберігати складні структури в атрибутах, займає менше місця і прямо інтерпретується за допомогою JavaScript в об'єкти. Наприклад:

```
{
  "firstName": "Іван",
  "lastName": "Коваленко",
  "address": {
    "streetAddress": "вул. Грушевського 14, кв.101",
    "city": "Київ",
    "postalCode": 21000
```

```
  },
  "phoneNumbers": [
    "044 123-1234",
    "050 123-4567"
  ]
}
```

Технічно REST-сервіси можна реалізувати за допомогою статичних HTML-сторінок.

Якщо REST-запит викличе на виконання віддалений створюваний код, то повертається XML-відгук, проте даний факт у REST замаскований краще, ніж у його SOAP-конкурента, який обов'язково потребує вказівки тієї функції, яка буде виконана.

REST популярна завдяки тому, що хостування REST-сервісів − відносно простий процес у порівнянні із зусиллями, необхідними для розгортання інфраструктури та коду з метою підтримки SOAP-комунікацій. Ще в 2003 р. Джеф Бар, пропагандист Web-сервісів з Amazon.com, розповів, що його компанія обробляє більше REST-, ніж SOAP-запитів.

Слід також враховувати, що майже всі канали RSS і ATOM також являються REST-сервісами. Власне, будь-який URI-запит, який передбачає звичайну XML-відповідь (POX - Plain-Old XML), технічно залишається REST-сервісом.

Переваги та недоліки.

Пропагандисти REST-сервісів стверджують, що вони краще масштабуються і, отже, краще справляються з великими обсягами обслуговування. SOAP-сервіси набагато сильніше завантажують обчислювальні ресурси, оскільки вимагають розбору XML-коду та впорядкування об'єктів, - а для оптимізації управління та прискорення цих операцій необхідно спеціальне програмне та апаратне забезпечення. Тоді як REST-сервіси є HTTP-запитами, а отже легко контролюються штатними засобами вирівнювання навантаження звичайних пристроїв керування, наприклад, від компаній Cisco Systems,

Citrix і F5 Net-works. Моніторинг використання REST-сервісів теж значно простіше і часто менш дорогий, оскільки моніторинг на базі URI вже став сформованою технологією; пропонуються як відповідні комерційні продукти, так і рішення з відкритим вихідним кодом.

REST-сервіси простіше реалізувати, оскільки вони базуються на добре відомих Web-протоколах і не вимагають від розробника вивчення WSDL, SOAP та інших WS-специфікацій, що використовуються для управління і забезпечення безпеки SOAP-сервісів. Найпростіший REST-сервіс можна реалізувати за кілька хвилин - статичний XML-файл, що повертається Web-сервісом, це технічно і є REST-сервіс, адже XML-дані запрошувалися через HTTP. Це навряд чи може стати оптимальним способом побудови сервісної інфраструктури, але для статичних або рідко мінливих ресурсів така можливість вельми приваблива.

Але не все досконало у світі REST. Оскільки технологія REST базується на HTTP, то вона несе на собі відбиток ненадійності цього протоколу і неможливість збереження стану (його stateless-характеру) на відміну від складних WS-специфікацій.

REST також не забезпечує стандартизованого механізму доступу до ресурсів, отже, самостійно доведеться визначати такі стандарти і забезпечувати їх виконання, а також інформувати про спосіб доступу до ресурсів сервісів своїх партнерів, розробників і користувачів. Необхідно своєчасно оновлювати цей механізм комунікацій у разі будь-яких змін сервісу. У світі ж SOAP це виконує WSDL, діючи як динамічний контрагент між сервісом і його клієнтами.

### 8.4. Розробка web-сервісів з використанням .NET Framework

.NET Framework є платформою для створення web-додатків і web-сервісів, котрі працюють під управлінням Internet Information Server (IIS). Реалізація web-служб .NET здійснюється так само, як активізація віддаленої web-служби, або виклик методу локального класу. При цьому інструмента-

льні засоби, які надаються системою .NET Framework, дозволяють не знайомитись з роботою таких стандартів, як SOAP і WSDL.

Для створення web-служби необхідно виконати наступні дії (п. 1, 3, 5 і 8 виконуються вручну).

1. Розробляється web-служба як .NET-клас з атрибутами, які ідентифікують його як web-службу з деякими функціями.

2. У середовищі Visual Studio .NET автоматично створюється документ WSDL, де описується, як клієнт повинен взаємодіяти з web-службою.

3. Споживач (клієнт) знаходить web-службу і, вирішивши скористатися нею, додає відповідне web-посилання в проект Visual Studio .NET (або запускає утиліту wsdl.exe).

4. У середовищі Visual Studio .NET здійснюється автоматична перевірка документа WSDL і генерується проксі-клас, який дозволяє споживачу взаємодіяти з web-службою.

5. Споживач викликає один з методів класу web-служби. Для нього цей виклик не відрізняється від виклику методу будь-якого іншого класу, але насправді споживач взаємодіє з проксі-класом, а не з web-службою.

6. Проксі-клас перетворює передані параметри в повідомлення SOAP і відправляє його web-службі.

7. Незабаром проксі-клас одержує SOAP-відповідь, перетворює її у відповідний тип даних і повертає його як звичний тип даних .NET.

8. Споживач використовує повернену йому інформацію.

При роботі web-служб .NET використовується технологія ASP.NET, яка є частиною системи .NET Framework. Ця технологія вимагає підтримки з боку серверу Microsoft IIS. Середовище Visual Studio .NET забезпечує велику кількість інструментів, які допомагають полегшити вирішення задач, пов'язаних з отриманням і виконанням web-служби.

Робота web-служб побудована на використовуванні різних відкритих стандартів: WSDL, SOAP, HTTP; специфікації DISCO і UDDI, які полегшують публікацію і пошук інформації щодо web-служби (на сьогоднішній день

використовується також HTTP-комунікація і немає сенсу від неї відмовлятися); з застосуванням розширення WS-Inspection - специфікації для пошуку документів, в яких перераховані групи web-служб і їх місцезнаходження (розроблена компаніями Microsoft і IBM для заміни протоколу DISCO).

Крім того, існують конкуруючі специфікації, котрі служать для подолання деяких властивих web-службам обмежень, до яких можна віднести відсутність транзакцій, аутентифікації, ліцензування і шифрування. Жодна з подібних специфікацій не досягла рівня встановленого стандартом і не була включена в .NET, але, можливо, в майбутньому це відбудеться.

### 8.4.1. Атрибут WebService

Щоб повідомити клієнта про можливості, які надаються службою, потрібен відповідний механізм документування такої інформації. Середовище .NET для цього надає атрибут WebServiceAttribute, котрий є членом простору імен System.Web.Services. З його допомогою можна вказати клієнту, де знайти інформацію про web-службу. Як і атрибути інших типів, клас WebServiceAttribute успадковує функції класу System.Attribute. В більшості випадків для зручності використання слово Attribute в імені класу можна опустити, вказавши просто WebService. Для спрощення тексту програми використаємо дану можливість. Атрибут WebService має дві властивості:

- Namespace - встановлює для служби простір імен XML;
- Description - додає текстовий і HTML-опис служби, який стає частиною WSDL-документа опису служби.

Простори імен XML, необхідні для забезпечення унікальної ідентифікації елементів і атрибутів XML-документів, не мають нічого спільного з просторами імен .NET, які служать для організації класів.

У лістингу наведено документ web-служби DNSLookUpService, який використовує властивості Namespace і Description для реалізації атрибуту WebService. Цей документ містить хост-ім'я, як рядковий аргумент і застосо-

вує клас DNS з простору імен System.Net для визначення і повернення відповідної IP-адреси.

Документ web-служби DNSLookUpService

```
public class WebService : MarshalByValueComponent
{
using System.Web.Services;
using Sestem.Net;
[WebService(Namespace = "http://www.bostontechnical.com/webservices/"
Description="<P>Web-служба. яка виконує проглядання доменних імен.</P>")]
{
   [WebMethod]
   public string getIPforHostname(string strHostname)
   {
      IPHostEntry hostlnfo = Dns.GetHostByName(strHostname);
      return hostlnfo.AddressList[0].ToString();
   }
}
}
```

У наведеному прикладі простір імен для web-служби знаходиться за адресою. Web-служба використовуватиме цей простір як сховище всіх XML-документів, які повертаються у відповідь на її виклик. Простори імен XML призначені для виключення суперечностей в назвах елементів і атрибутів. Звичайно використовується простір імен URI як більш універсальний, оскільки він гарантує унікальність доменного імені (у нашому випадку - www.bostontechnical.com). Цей простір також указує область відповідальності організації (в даному випадку власника домена www.bostontechnical.com) за унікальність елементів домена в просторі імен /webservices.

### 8.4.2. Атрибут WebMethod

Атрибут WebMethod повідомляє середовище .NET, що деякий загальний метод повинен бути представлений як метод, що викликається через Web. Для документування і зміни поведінки web-методу даний атрибут включає наступні шість властивостей:

- Description - опис;
- MessageName – ім’я повідомлення;
- EnableSession – включений сеанс;
- CacheDuration – довготривалість кешування;
- TransactionOption – варіант угоди;
- BufferResponse – відповідь буфера.

Перші 2 властивості − для документування web-методу, тоді як інші визначають його поведінку.

Властивості Description і MessageName.

Необхідно включати опис в кожен web-метод, щоб споживачі не з’ясовували як працює web-служба. Це необхідно, зокрема, коли web-служба містить перевантажені методи (методи, з одним і тим же ім'ям, але з різними сигнатурами). Наприклад, в наступному лістингу оголошуються два методи з ім'ям Add(), один з яких приймає параметри типа Integer, а інший − параметри типа Float.

Опис методу

```
 [WebMethod]
   public int Add(int a, int b)
   {
      return a+b:
   }
[WebMethod]
   public float Add(float a. float b)
   {
```

```
        return a+b;
    }
```

При доступі до web-служби, яка містить два методи з одним і тим же ім'ям, але з різними сигнатурами (перевантаження), через тестову сторінку Internet Explore, при відкритті цієї сторінки виникне помилка часу виконання з повідомленням «Два методи, Single Add(Single. Single) і Int32 Add( Int32. Int32), використовують ім'я "Add". Зверніться до властивості MessageName атрибуту WebMethod для зазначення унікальних імен методів».

Процедура коментування web-методів аналогічна процедурі коментування служби. Кожен метод починається з атрибуту WebMethod. Для додавання опису web-методу використовується властивість Description цього атрибуту, а для зміни його імені - властивість MessageName. Наприклад

```
[WebMethod(MessageName="имя". Description="описание")]
```

### 8.4.3. Властивість EnableSession

Оскільки web-служби ASP.NET (класи одержані з простору імен System.Web.Services.) існують в контексті додатку ASP.NET, вони мають доступ до об'єктів Application і Session цього додатку. Додаток ASP.NET включає тільки один об'єкт Application, отже, він здатен підтримувати безліч об'єктів сеансу, призначених для зберігання даних різних клієнтів. За умовчанням такий механізм управління знаходиться в стані − відключений; активізувати його можна шляхом привласнення властивості EnableSession значення true. Використовування даного механізму може знизити продуктивність, тому включати його слід лише при необхідності.

Якщо механізм управління сеансами запущено, сервер управляє станом клієнтів за допомогою унікальних об'єктів HttpSessionState. При першій взаємодії користувача з сервером кожному об'єкту Session призначається унікальний ідентифікатор. У подальшому клієнт повинен надати серверу свій унікальний ідентифікатор для одержання всіх клієнтських даних, збережених в попередньому сеансі. Унікальний ідентифікатор може зберігатися в cookie-

файлі на комп'ютері клієнта або поміщатися в URL-адресу. Для доступного через браузер додатка ASP.NET така система управління станом працює у фоновому режимі. Якщо в web-браузер включена підтримка cookie-файлів, вони автоматично надаватимуться серверу по відповідному запиту. Оскільки веб-служби не можуть одержувати cookie-файли аналогічним чином, то при кожному виклику web-служби необхідно указувати cookie-файл програмно.

Виклик cookie-файлу

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.serviceModel>
<bindings>
<basicHttpBinding>
<binding name="AdministrationServiceSoap" closeTimeout="00:01:00"
 openTimeout="00:01:00" receiveTimeout="00:10:00"
 sendTimeout="00:01:00"
 allowCookies="true" bypassProxyOnLocal="false"
 hostNameComparisonMode="StrongWildcard"
                …
 </binding>
 </basicHttpBinding>
 </bindings>
 <client>
 <endpoint
address="http://localhost/AdministrationWebService/AdministrationService.asmx"
  binding="basicHttpBinding"                                    bindi-
?gConfiguration="AdministrationServiceSoap"
contract="AdministrationServiceProxy.AdministrationServiceSoap"
   name="AdministrationServiceSoap" />
  </client>
  </system.serviceModel>
```

```
    </configuration>
    Session timeout
    <system.web>
  <sessionState mode="InProc" cookieless="false"
timeout="1"></sessionState>
  …
  </system.webServer>
```

За замовчуванням, сесія буде створена у процесі де web-сайт працює (InProc). Це контролюється за допомогою параметра у файлі web.config:

```
  <sessionState mode="InProc" />
```

Працювати з сесією в режимі поточного часу досить зручно, але це означає, що стан сесії буде втрачено при його завершенні (наприклад, при оновленні). Є альтернативні режими використання, які дозволяють зберегти стан сеансу, навіть якщо додаток переробляється. Доступні варіанти:

- Off – Стан сесії не зберігається;

- InProc (за замовчуванням) - стан сесії існує в рамках процесу мережі користувача;

-StateServer - дані сесії передаються в настройках послуги StateServer;

-SQLServer - дані сесії зберігаються в налаштуваннях бази даних SQL Server.

Обидва режими StateServer та SQLServer дозволяють зберігати стан сеансу. Але при збереженні посилання типу об'єкт (таких як клас випадків), вони можуть зберігатися, якщо позначені атрибутом Serializable.

За замовчуванням, якщо користувач не має доступу до своєї сесії протягом 20 хвилин, вона закінчується, і все що зберігалося буде втрачено. У зв'язку з цим, важливою є перевірка об'єкту, який повернувся з сесії, щоб побачити, чи є він недійсним, перш ніж намагатися працювати з ним. Ця перевірка може бути виконана наступним чином:

```
object sessionObject = Session["someObject"];
if (sessionObject != null) {
 myLabel.Text = sessionObject.ToString();
}
```

Час сеансу регулюється за допомогою файлу Web.config.

```
<sessionState timeout="number of minutes" />,
```

але зростання його значення (timeout) може призвести до небажаного перевантаження пам'яті на сервері. Інформація сесії впродовж заданого часу (timeout) в ASP.NET зберігається у вигляді хеш-таблиці − структури даних, що реалізує інтерфейс асоціативного масиву, а саме, вона дозволяє зберігати пари (ключ, значення) і здійснювати три операції: додавання нової пари, пошуку і видалення за ключем. Наприклад, присвоєння змінним "username" і "color" значення "Aayush Puri" і "Blue" виконується наступним чином, відповідно:

```
Session(“username”) = “Aayush Puri”
Session(“color”) = “Blue”
```

Якщо в подальшому потрібно отримати значення "username" або "color" з сесії використовується функція Session("username") та Session("color").

Сесії в ASP.NET представлені за допомогою 32-розрядних цілих чисел, що визначають ідентифікатори сеансу. Двигун ASP генерує ці ідентифікатори сесій так, що унікальність гарантується. Розглянемо процедури налаштування об'єкта сесії (табл.18).

Налаштування об'єкту сесії в залежності до вимог до веб-додатку

Таблиця 18

| Тип Session | Дія | Приклад |
|---|---|---|
| Session.Abandon | Відміна поточної сесії | |
| Session.Remove | Видалення елемента з сесії | Session(“username”) = “Aayush Puri” (Ініціація змінної сесії) Session.Remove(“username”) (Ви- |

| | | |
|---|---|---|
| | | даляє змінну сесії “username”) |
| Session.RemoveAll | Видаляє всі елементи сесії. | |
| Session.Timeout | Встановлює тайм-аут (у хвилинах) для сесії | Session.Timeout=30 (Якщо користувач не запрошує сторінку упродовж 30 хвилин, сесія завершується) |
| Session.SessionID | Отримує ідентифікатор сеансу (тільки для читання) для поточної сесії. | |
| Session.IsNewSession | Це перевірка чи уперше користувач звернувся до цієї сторінки ASP.NET. | |

### 8.4.4. Налаштування SQL Server для ASP.NET

Розглянемо, як скористатися SQL-сценаріями InstallSqlState.sql та UninstallSqlState.sql для налаштування SQL Server у режимі стану сесії управління. За замовчуванням вони знаходяться в папці:

<Шлях до директорії Windows>\Microsoft.NET\Framework\<Шлях до версії .NET Framework>

Для відкриття файлів сценаріїв в оболонці **SQL Query Analyzer** необхідно вибрати пункт **Open** опції меню **File,** та вказати шлях до файлів сценарію в діалоговому вікні **Open Query File**, та натиснути кнопку **Open**.

Для виконання сценарію в **SQL Query Analyzer**, необхідно вибрати опцію **Execute (Виконати)** у меню **Query (Запит)**.

Перед запуском сценарію UninstallSqlState.sql для видалення SQL Server Session State, необхідно зупинити **w3svc** процес. Для цього виконуються наступні кроки:

1. У меню **Пуск** виберіть пункт **Виконати** (**Run**), введіть команду **CMD** та натисніть кнопку **OK**, щоб відкрити вікно командного рядка.

2. В командному рядку введіть **net stop w3svc**. Отримаємо підтвердження, що **w3svc** процес зупинився.

3. У SQL Query Analyzer, у меню **Файл** (**File**) виберіть пункт **Відкрити** (**Open**).

4. У діалоговому вікні **Відкрити (Open) → Запрос (Query) → Файл** (**File**) знайдіть сценарій UninstallSqlState.sql, натисніть кнопку **Відкрити** (**Open**) (відкривається в SQL Query Analyzerта).

5. Запустіть сценарій UninstallSqlState.sql, для цього виберіть пункт **Виконати** (**Execute**) в меню **Запит** (**Query**).

6. Перезавантажте **w3svc** служби. Щоб перезапустити **w3svc** процес, виконайте команду «**net start w3svc»** у командному рядку cmd.

**Зміна Web.config файлу програми**

Для виконання ASP.NET SQL Server в режимі управління станом сеансу, необхідно змінити елемент файлу Web.config <**sessionState**> наступним чином.

Встановіть атрибут <sessionState mode=”  ”> елемент SQLServer для зазначення, що стан сеансу зберігається в SQL Server.

Встановіть атрибут sqlConnectionString, вказавши рядок з'єднання для SQL Server. Наприклад:

sqlConnect-

?nect-

?onString="data source=MySQLServer;user id=<username>;password=<strongpassword>"

*Примітка.* Користувач <username> повинен мати дозвіл на виконання цієї операції у базі даних.

Змінений <sessionState> елемент має виглядати наступним чином:

```
<sessionState mode="SQLServer" sqlConnectionString="data source=127.0.0.1;user id=<username>;password=<strongpassword>" cooki-
?less="false" timeout="20" />
```

Цей код вводиться з врахуванням регістру.

### 8.4.5. Властивість CacheDuration

Завдяки використанню кешування в web-службах підвищується їх масштабованість і продуктивність. Для цього застосовують властивість CacheDuration атрибуту WebMethod. Такий тип кешування називається кешуванням результатів і в середовищі .NET виконується шляхом збереження на певний час відповідних пар запитів і відповідей в кеш-таблиці, яка знаходиться в пам'яті. Протягом цього часу на запит, який вже поступив та співпадає з вже існуючим в кеші, сервер видає кешовану відповідь. При використанні CacheDuration задають кількість секунд (значення цілого типу), впродовж яких пара запит-відповідь зберігатиметься в кеші. За умовчанням це значення дорівнює 0, тобто відповіді не кешуються.

Механізм кешування виявляється ідеальним засобом для web-методів, які обробляють запити інтенсивно використовуючи процесор, або інші складні запити, що вимагають незначної зміни раніш одержаних результатів. До числа подібних web-методів відноситься метод, який запитує з бази даних щодня змінні заголовки новин. Для зниження кількості звертань до бази даних можна встановити параметр часу CacheDuration, що дорівнює 5 або більш хвилинам. Оскільки система кешування ґрунтується на парі запит-відповідь, то у разі, коли діапазон вхідних параметрів методу невеликий, певні ресурси серверу використовуються в багатьох аналогічних ситуаціях. Якщо діапазон вхідних параметрів надто великий (тобто запит включає велику кількість рядків), кеш-таблиця може дуже вирости та займати великий об'єм пам'яті, пошук необхідних даних буде займати багато часу. Ситуація ускладнюється, якщо об'єм вихідних даних, що зберігаються в кеші, також великий.

Встановлення параметра рівним 1 хвилину

```
[WebMethod(Description = "Number of times this service has been accessed", CacheDuration = 60, MessageName = "ServiceUsage")]
public int ServiceUsage()
```

```
        {
            // If the XML Web service has not been accessed, initialize it to 1.
            if (Application["MyServiceUsage"] == null)
            {
                Application["MyServiceUsage"] = 1;
            }
            else
            {
                // Increment the usage count.
                Application["MyServiceUsage"] =
((int)Application["MyServiceUsage"]) + 1;
            }
            // Return the usage count.
            return (int)Application["MyServiceUsage"];
        }
```

### 8.4.6. Властивість TransactionOption

Транзакції можна представити як набір процедур (подій, викликів функцій тощо), виконання яких приводить до певної зміни стану (як у разі успішного завершення необхідних процедур, так і у разі збою однієї з них). Як приклад можна привести систему обробки кредитних карток, яка аутентифікує номер кредитної картки, знімає з рахунку гроші (при їх наявності) і здійснює фіксацію транзакції. Кожна з цих трьох операцій є частиною транзакції. Якщо на одному з етапів відбувається збій (наприклад, картка недійсна), вся транзакція відміняється, а змінені дані відновлюються.

Компанія Microsoft включила підтримку транзакцій, подібних транзакціям MTS і COM+, в платформу .NET за допомогою простору імен System.EnterpriseServices. Підтримка транзакцій .NET задається властивістю TransactionOption атрибуту WebMethod. Ця властивість може мати одне з наступних п'яти значень:

- Disabled (вимкнена);
- NotSupported (не підтримується);
- Supported (підтримується);
- Required (потребує);
- RequiresNew (потребує нових).

За замовчанням транзакції відключені. Якщо ж потрібно використовувати .NET-транзакції, тоді web-метод братиме участь в транзакціях тільки як кореневий об'єкт. Це означає, що метод зможе викликати інші доступні для транзакцій об'єкти, але його не можна буде викликати як частину транзакції, початої іншим об'єктом. Дане обмеження обумовлене тим, що протокол HTTP за своєю природою є stateless-протоколом, тобто не зберігає попередній стан. В результаті значення Required і RequiredNew властивості Trans-?ctionOption виявляються еквівалентними і оголошують метод RequiredNew, який починає нову транзакцію. Значення, що залишилися - Disabled, NotSupported і Supported - відключають транзакції для web-методу. Як ілюстрацію сказаного розглянемо наступний приклад:

```
alter Procedure AddUser
  (
    @login nvarchar(50),
    @password nvarchar(50),
    @active int,
    @usertypeid int,
    @id int out
  )
as
  set nocount on
  insert into [user]
    ([login], [password], [usertypeid],  [active] )
  values
    (@login,@password, @usertypeid, @active)
```

```
  select @id = @@identity
GO
```

Приклад роботи транзакцій: додавання користувачів – всі або ніхто

```
    [WebMethod(false, TransactionOption.Required)]
    public int AddUser(String login, String password, int active)
    {
       try
       {
          int Id = userManager.AddUser(login, password, active);
          return Id;
       }
       catch (Exception ex)
       {
          throw new Exception(ex.Message, ex);
       }
    }
    [WebMethod(false, TransactionOption.Required)]
    public bool AddUserCollection(List<User> userCollection)
    {
       try
       {
          foreach(User user in userCollection)
          {
             AddUser(user.Login, user.Password, user.Active);
          }
          return true;
       }
       catch (Exception e)
```

```
        {
            throw e;
        }
    }
```

### 8.4.7. Властивість BufferedResponse

За замовчанням передбачається, що web-метод зберігає відповідь в буфері пам'яті до тих пір, доки відповідь не завершиться або буфер не заповниться. Таке зберігання відповіді називається серіалізацією (serialization). В більшості випадків при серіалізації за рахунок зменшення кількості сеансів зв'язку з клієнтом підвищується продуктивність системи. Але якщо web-метод повертає великий об'єм даних або для його виконання потрібна значна кількість часу, то можна відключити буферізацію шляхом привласнення властивості BufferedResponse значення false. Таке значення наказує середовищу .NET після серіалізації відповіді відправляти її клієнту, проте при невеликих наборах даних це може привести до зниження продуктивності.

Іноді має сенс відключити буферизацію для тривало виконуваних aspx-сторінок (наприклад, таких як пошукові сторінки), вихідні дані яких відображуються на екрані, щоб користувач міг почати перегляд інформації до її повного отримання.

Кешування. Правильно виконане кешування може значно підвищити продуктивність програми. Для цього потрібно завчасне планування і невеликий за об'ємом код. Стосовно web-служб кешування дозволяє забезпечити ефективніше працюючий з погляду серверу код.

Кешування реалізується у вигляді окремої служби, яка є частиною ASP .NET. Існують два типи кешування, які може використовувати у web-службі – кешування результатів і кешування даних.

Кешування результатів – це збереження одержаних після виклику web-методу результатів і повторне їх використання іншими клієнтами, які відправили в певний відрізок часу запит з тими ж параметрами. Кешування ре-

зультатів (вихідних даних), якщо вони вже одержані, не вимагає для своєї реалізації коду в web-службі і виконується автоматично.

Просте кешування результатів. Для підключення режиму простого кешування достатньо задати для атрибуту WebMethod значення властивості CacheDuration. Час кешування результатів завжди указується в секундах і відноситься до конкретного методу.

Метод, що повертає поточний час, є прикладом демонстрації кешування результатів. У наведеному нижче фрагменті при першому виклику методу виконується його код, який показує поточний час. Якщо наступний виклик коду відбудеться менш ніж через 60 секунд після першого, автоматично буде повернений кешований результат:

```
[WebMethod(CacheDuration=60)]
  public string GetDateTime()
  {
     return System.DateTime.Now.ToString();
  }
```

Середовище ASP.NET автоматично управляє процесом кешування. Однією з найкорисніших властивостей кешування результатів web-служби являється те, що відповідь використовується повторно тільки коли список представлених методу параметрів виявляється тим самим.

Кешування результатів можна також включити шляхом додавання директиви <%@ OutputCache %> у asmx-частину web-служби. Проте не рекомендується застосовувати вказаний метод, оскільки web-служба підтримує не всі стандартні атрибути директиви OutputCache. Крім того, Visual Studio .NET не дозволяє безпосередньо модифікувати asmx-файл.

Кешування даних в web-сервісах. На відміну від кешування результатів, при якому зберігаються тільки результати роботи методів web-служби, кешування даних дозволяє зберігати інформацію будь-якого типу. Можна помістити потрібні дані в кеш, а потім у будь-який момент витягнути їх звідти. Недолік цього − необхідність створення в web-методі відповідного програмного коду.

Основою для кешування даних служить проста *словарна колекція Cache* (екземпляр класу System.Web.Caching.Cache). Колекція Cache індексує кожен елемент з використанням описового рядка і здатна прийняти будь-який об'єкт, зокрема призначені для користувача дані і бізнес-логіку. Колекції Cache і Application мають дві фундаментальні відмінності.

Об'єкт Cache являється захищеним на рівні потоку. А це означає, що при додаванні і видаленні елементів не потрібно використовувати методи Lock() і UnLock(). Але проблема конкуренції за доступ до об'єкту існуватиме, якщо в кеші знаходиться об'єкт, який не є потоко-захищеним, а змінити або використати його намагатимуться декілька клієнтів. Цю проблему можна обійти або шляхом створення в кеші тимчасової копії об'єкту, або використовуючи копію в web-методі.

Для доступу до колекції Cache в веб-службі слід застосовувати властивість Context.Cache, якщо веб-служба належить класу System.Web.Services.WebService, і властивість HttpContent.Current.Cache в протилежному випадку.

Додавання елементів в кеш. Для додавання в колекцію Cache елемента треба присвоїти йому нове ключове ім’я: Context.Cache["key"] = dsProducts.

Проте в даному випадку відсутня можливість визначити політику закінчення часу зберігання об'єкту в кеші, внаслідок цього для додавання елементу в колекцію Cache звичайно використовують методи Insert(). Існує чотири такі методи.

1) Context.Cache.Insert(key, value). Поміщає елемент в кеш під вказаним ключовим ім'ям і призначає заданий за замовчуванням пріоритет і час зберігання. Це тотожно використовуванню індексованої колекції і привласненню нового ключового імені.

2) Context.Cache.Insert(key, value, dependencies). Поміщає елемент в кеш під вказаним ключовим ім'ям і з використанням заданих за замовчанням пріоритету і часу зберігання. Останній параметр містить об'єкт Cache-

?ependency, який зв'язує файли або кешовані елементи і забезпечує видалення кешованих елементів при їх зміні.

3) Context.Cache.Insert(key, value, dependencies, absoluteExpiration, slidingExpiration).

Поміщає елемент в кеш під вказаним ключовим ім'ям, використовуючи заданий за замовчанням пріоритет і заданий абсолютний або плавучий час зберігання (не можливо встановити відразу обидва значення часу). Це найчастіше використовувана версія методу Insert().

4) Context.Cache.Insert (key, value, dependebcies, absoluteExpiration, slidingExpiration, Priority onRemoveCallback).

Дозволяє конфігурувати всі варіанти кешування для елементу, включаючи закінчення терміну зберігання, залежності і пріоритет. Можна застосувати делегат, щоб задати метод, який викликається при видаленні елементу.

Установка web-серверу IIS. Для роботи з Web-сервісами необхідно установити і налаштувати Web сервер. Для цього використовується Web- сервер, який є частиною Windows – IIS.

IIS (Internet Information Services) — це набір серверів для служб Інтернету від компанії Майкрософт. IIS поширюється з операційними системами родини Windows NT. Основний компонент IIS – web-сервер, який дозволяє розміщувати в Інтернеті сайти. IIS підтримує протоколи HTTP, HTTPS, FTP, POP3, SMTP, NNTP.

Для встановлення IIS, необхідно вибрати «Пуск / Панель управління / Установка і видалення програм / Установка компонентів Windows». В отриманому вікні вибрати Internet Information Services (IIS).

Щоб відкрити вікно налаштування Web-серверу необхідно здійснити наступні кроки «Пуск / Панель управління / Адміністрування / Internet Information Services». Деталі встановлення IIS можна знайти в [IIS].

Синхронні http-обробники. Наведемо приклад простого синхронного обробника http.

Sync.ashx, синхронні http-обробники

```
<%@ WebHandler Language="C#" Class="AsyncHttpHandler"%>
using System;
using System.Web;
public class AsyncHttpHandler : IHttpHandler {
public void ProcessRequest(HttpContext context) {
context.Response.Write("<H1>This is an Http handler.</H1>");
  }
  public bool IsReusable { get { return true; }  }
}
```

Скопіюйте текст з лістингу 1 в файл з розширенням .ashx і перенесіть його у віртуальний кореневий каталог для IIS (на машині, де встановлений .NET Framework). Тепер необхідно звернутися до цього файлу через браузер. ASP.Net при компіляції коду в перший раз запрошує ресурс. Кожен наступний раз, коли файл запрошується, ProcessRequest() викликається метод для отримання http-відповіді, що доноситься до клієнта. Код з останнього лістингу є одним з основних обробників HTTP, гнучка функція ASP.NET дійсна (і простіше, ніж ISAPI.DLL). Основний недолік − не може бути викликана асинхронно ASP.NET.

Розглянемо, чому синхронні обробники можуть бути менш досконалі.

Найбільш активним кодом серверу являється код, що відповідає за введення/виведення, а не код обчислення на CPU. Це означає, що вузьке місце у формуванні відповіді, як правило, знаходиться близько  до другого рівня комунікаційної мережі, тобто бази даних або файлової системи. Коли IO-зв’язаний http-обробник має велике навантаження, то його потік простоює в очікуванні завершення процесу вводу/виводу. А для початку роботи другого клієнту створюється ще один потік, це не є ефективним. Це недолік синхронних обробників http-запитів.

Асинхронні обробники http. Під час роботи асинхронного обробника http-даних ASP.NET переміщує потік, який використовується для зовнішнього процесу, назад в пул потоку до того часу, поки обробник не отримає зворотний виклик із зовнішнього процесу. Це дозволяє запобігти блокуванню потоку і підвищує продуктивність, оскільки одночасно може виконуватися тільки обмежене число потоків. Якщо кілька користувачів запитають синхронні обробники http-даних, що залежать від зовнішніх процесів, в операційній системі можуть швидко закінчитися вільні потоки, оскільки багато з них будуть знаходитися в стані блокування або очікування зовнішнього процесу.

Для реалізації асинхронного обробника http, ваш клас повинен наслідувати інтерфейс IHttpAsyncHandler, а не інтерфейс IHttpHandler. Але інтерфейс – це не єдина відмінність між синхронними та асинхронними обробниками http. Конструкція програми стає трохи складнішою. Ця складність полягає у тому, що потрібно реалізувати асинхронну програмну модель. Усі запити вводу-виводу, які виконуються під час процесу формування http-відповідей, повинні бути асинхронними, використовуючи методи Begin*() та End*() замість їх синхронних еквівалентів. Наприклад, якщо обробнику треба прочитати файл, він повинен викликати методи BeginRead()/EndRead() об’єкту Stream, а не синхронний еквівалент цих методів - Read(). Метод BeginRead() викличе код після того, як закінчить читання. Тим часом потік може бути повернений до пулу, щоб він не простоював.

У додаток до безпосереднього використання асинхронних методів Begin*() та End*(), асинхронний обробник повинен реалізовувати асинхронні методи BeginProcessRequest() та EndProcessRequest(), що викликаються теж асинхронно самою ASP.NET. Методи BeginProcessRequest() та EndPr?cessRequest() належать до інтерфейсу IhttpAsyncHandler, що дозволяє середовищу ASP.NET виконувати асинхронно те, що воно робило синхронно за допомогою методу ProcessRequest() інтерфейсу IhttpHandler.

Створюючи асинхронний http-обробник необхідно пам’ятати наступне.

1) Усі запити вводу-виводу повинні проводитися за допомогою відповідних методів Begin*() та End*().

2) Необхідно реалізувати методи BeginProcessRequest() та EndPr-?cessRequest() в програмі, щоб ASP.NET змогло звертатися до коду асинхронно.

3) Потрібно створити тип даних IAsyncResult, об'єкт якого треба повернути до ASP.NET з методу Begin*(). ASP.NET перенаправляє цей об'єкт назад у метод End*(). Цей об'єкт потрібен для відстеження стану клієнтського запиту, так як процес поділено між декількома викликами.

Код з лістингу 18 демонструє простий асинхронний обробник HTTP.

Async.ashx, асинхронний обробник HTTP

```
<%@ WebHandler Language="C#" Class="AsyncHttpHandler"%>
using System;
using System.Web;
using System.Threading;
public class AsyncHttpHandler : IHttpAsyncHandler {
  public IAsyncResult BeginProcessRequest(
    HttpContext context, AsyncCallback cb, Object extraData){
    // Create an async object to return to caller
    Async async = new Async(cb, extraData);
    // store a little context for us to use later as well
    async.context = context;
    // Normally real work would be done here... then, most likely
    // as the result of an async callback, you eventually
    // "complete" the async operation so that the caller knows to
    // call the EndProcessRequest() method
    async.SetCompleted();
    // return IAsyncResult object to caller
    return async;
  }
```

```
  // Finish up
  public void EndProcessRequest(IAsyncResult result){
    // Finish things
    Async async = result as Async;
    async.context.Response.Write(
      "<H1>This is an <i>Asynchronous</i> response!!</H1>");
  }
  // This method is never called by ASP.Net in the async case
  public void ProcessRequest(HttpContext context) {
    throw new InvalidOperationException(
            "ASP.Net should never use this method");
  }
  // This means that the same AsyncHttpHandler object is used
  // for all requests returning false hear makes ASP.Net create
  //  object per request.
  public bool IsReusable {
    get { return true; }
  }
}
// This object is necessary for the caller to help your code keep
// track of state between begin and end calls
class Async:IAsyncResult{
  internal Async(AsyncCallback cb, Object extraData){
    this.cb = cb;
    asyncState = extraData;
    isCompleted = false;
  }
  private AsyncCallback cb = null;
  private Object asyncState;
  public object AsyncState {
```

```
  get {
    return asyncState;
  }
}
public bool CompletedSynchronously {
  get {
    return false;
  }
}
// If this object was not being used solely with ASP.Net this
// method would need an implementation. ASP.Net never uses the
// event, so it is not implemented here.
public WaitHandle AsyncWaitHandle {
  get {
    throw new InvalidOperationException(
            "ASP.Net should never use this property");
  }
}
private Boolean isCompleted;
public bool IsCompleted {
  get {
    return isCompleted;
  }
}
internal void SetCompleted(){
  isCompleted = true;
  if(cb != null){
     cb(this);
  }
}
```

```
  // state internal fields
  internal HttpContext context=null;
}
```

У майбутніх версіях .NET Framework, можливо з’явиться більше інновацій щодо асинхронної моделі програмування.

### 8.4.8. Оброблювач гіпертекстового транспортного протоколу

ASP.NET має маловідому особливість, яка дозволяє легко здійснювати процес „Оброблювач гіпертекстового транспортного протоколу”. Коли запит щодо сторінки надходить до ASP.NET, то цей запит обробляється об'єктом, який реалізує інтерфейс IHttpHandler. Цей інтерфейс включає метод "ProcessRequest", який відповідає за лист всього інформаційного наповнення сторінки до HttpContext.Response. Output stream. Файли ASHX дозволяють легко писати клас IHttpHandler, навіть не маючи можливості попередньо компілювати його.

Продемонструємо проект для відновлення зображення з баз даних SQL.

Необхідно створити файл у своєму проекті з розширенням ASHX. Назвати свою сторінку NWEmpPhoto.ashx. Відновити дані в таблиці Службовців бази даних Northwind. Відповідний вміст файлу ASHX має наступний вигляд :

```
<%@ webhandler language="C#" class="NWEmpPhotoHandler" %>
using System;
using System.Web;
using System.Data;
using System.Data.SqlClient;
public class NWEmpPhotoHandler : IHttpHandler
{
   public bool IsReusable { get { return true; } }
   public void ProcessRequest(HttpContext ctx)
   {
```

```
            string id = ctx.Request.QueryString["id"];
            SqlConnection con = new SqlConnection(<<INSERT CONNECTION
STRING HERE>>);
            SqlCommand cmd = new SqlCommand("SELECT Photo FROM
Employees WHERE EmployeeID = @EmpID", con);
            cmd.CommandType = CommandType.Text;
            cmd.Parameters.Add("@EmpID", id);
            con.Open();
            byte[] pict = (byte[])cmd.ExecuteScalar();
            con.Close();
            ctx.Response.ContentType = "image/bmp";
            ctx.Response.OutputStream.Write(pict, 78, pict.Length - 78);
        }
    }
```

В основному вихідному коді зі специфікатором <% webhandler%> клас реалізує два методи - IsReusable і ProcessRequest. IsReusable потрібен для обробки об'єднання. Метод ProcessRequest реалізує доступ до бази даних. Використовуємо ADO.NET для повернення зображення з бази даних. Зображення буде повернуте як масив байтів. Первинний ключ зображення передається в строку запитів. Використовуємо ExecuteScalar оскільки повертаємо один стовбець з однієї строки. Встановлюємо правильний тип даних (зображення / BMP в даному випадку) і формуємо відповідь у вигляді масиву байтів, що містять зображення на Response.OutputStream.

Для використання NW Photo Handler застосовуйте тег HTML-зображення. Оскільки первинний ключ передається в рядок запиту, необхідно включити його в атрибут SRC тегу HTML-зображення. Приклад тега зображення:

```
<img src="NWEmpPhoto.ashx?id=1" />
```

Універсальний обробник. «Універсальний» обробник додається до сайту стандартним шляхом. У провіднику рішення (Solution Explorer), слід натиснути праву кнопку миші на вузлі Веб-сайту CustomHandler і обрати Add New Item. Виберіть Generic Handler (рис. 59).

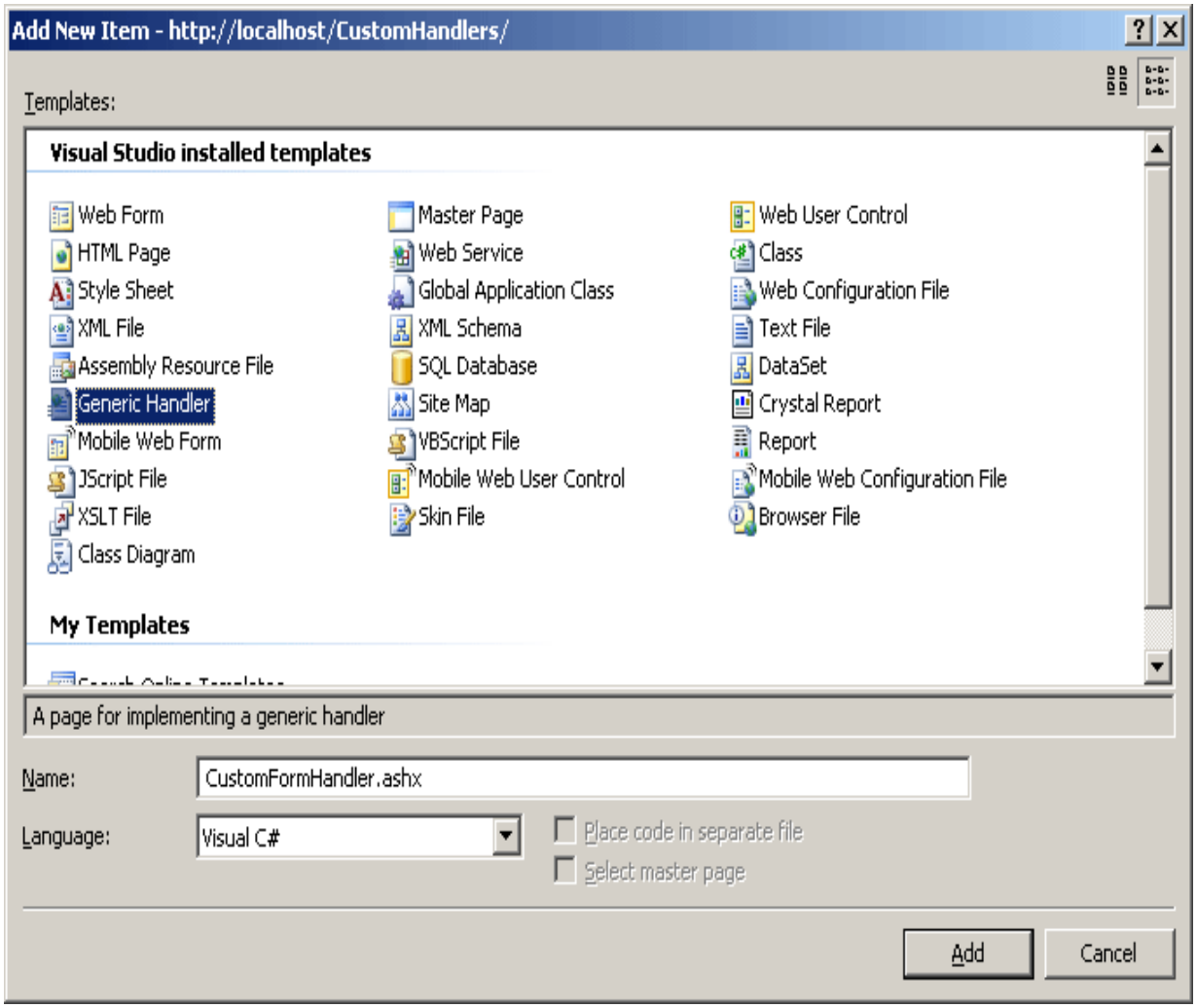



Рис. 59. Створення універсального оброблювача

Visual Studio генерує оброблювач CustomFormHandler.ashx, який включає метод ProcessRequest і властивість IsReusable.

```
<%@ WebHandler Language="C#" %>
using System.Web;
public class CustomFormHandler : IHttpHandler {
    public void ProcessRequest (HttpContext context) {
```

```
        context.Response.ContentType = "text/plain";
        context.Response.Write("Hello World");
    }
    public bool IsReusable {
        get {
            return false;
        }
    }
}
```

Беремо код з попереднього прикладу. Заміняємо метод ProcessRequest і властивість IsReusable  реальними реалізаціями.

```
<%@ WebHandler Language="C#" %>
using System.Web;
public class CustomFormHandler : IHttpHandler {
    public void ProcessRequest (HttpContext context) {
        ManageForm(context);
    }
    public void ManageForm(HttpContext context)
    {
        context.Response.Write("<html><body><form>");
        context.Response.Write(
          "<h2>Hello there. What's cool about .NET?</h2>");
        context.Response.Write("<select name='Feature'>");
        context.Response.Write("<option> Strong typing</option>");
        context.Response.Write("<option> Managed code</option>");
        context.Response.Write("<option> Language agnosticism
        </option>");
        context.Response.Write("<option> Better security model</option>");
        context.Response.Write(
```

```
            "<option> Threading and async delegates</option>");
        context.Response.Write("<option> XCOPY deployment</option>");
        context.Response.Write(
            "<option> Reasonable HTTP handling framework</option>");
        context.Response.Write("</select>");
        context.Response.Write("</br>");
       context.Response.Write(
            "<input type=submit name='Lookup' value='Lookup'></input>");
        context.Response.Write("</br>");
        if (context.Request.Params["Feature"] != null)
        {
            context.Response.Write("Hi, you picked: ");
            context.Response.Write(context.Request.Params["Feature"]);
            context.Response.Write(" as your favorite feature.</br>");
        }
        context.Response.Write("</form></body></html>");
    }
    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}
```

Якщо запустити CustomFormHandler.ashx на виконання, він повинен працювати так, як і обробники, які реалізовані в CustomFormHandler.cs і CustomFormHandler.vb, як показано на рис.60.

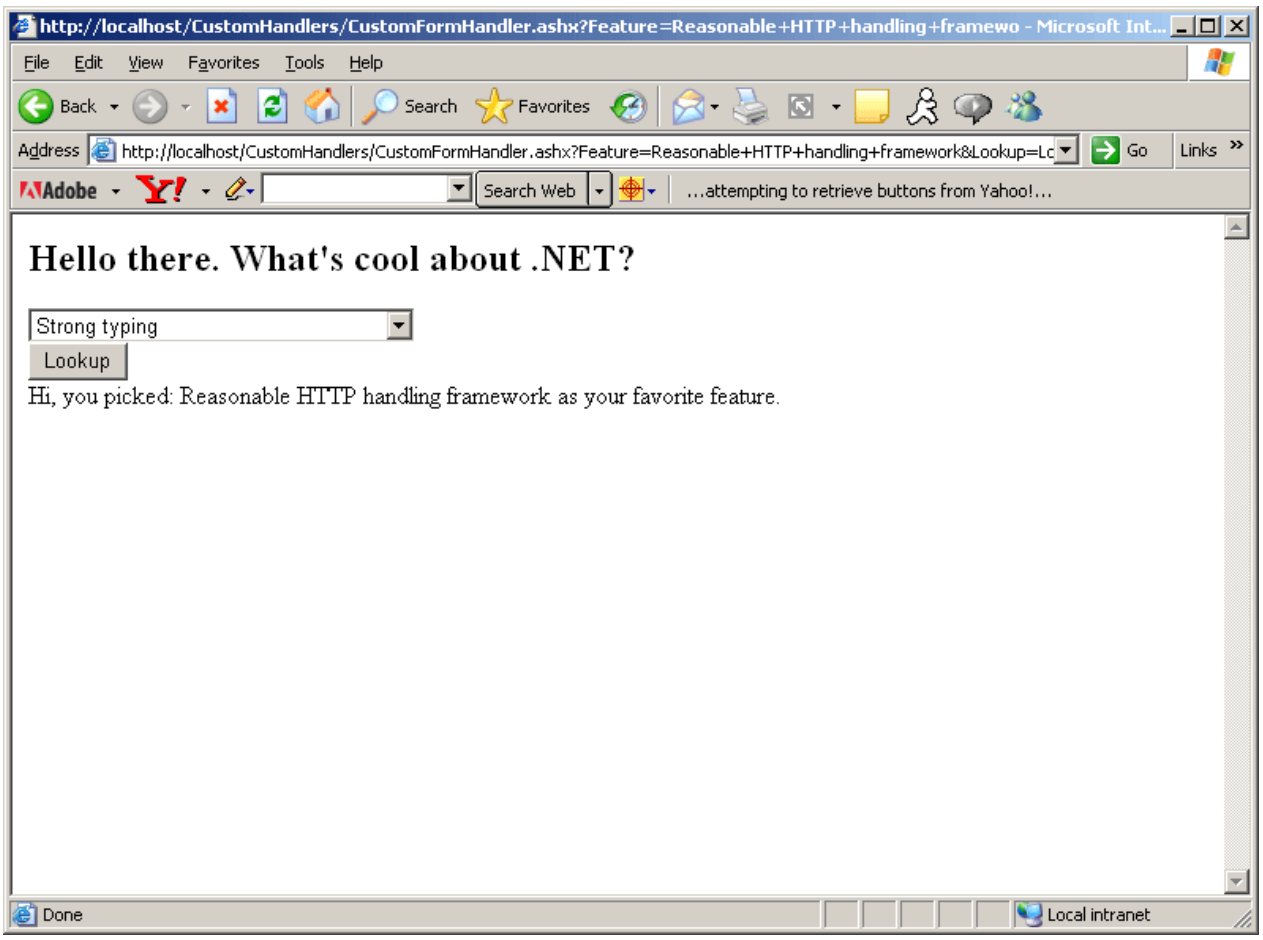



Рис. 60. Перегляд результату у браузері

Існує дві переваги використання універсального обробника. По-перше, як правило, простий оброблювач генерувати набагато зручніше, ніж створювати цілий новий сайт, щоб обробити запит. По-друге, не потрібно втручатися в Web.Config або IIS. Тобто Web.Config та IIS вже розуміють, що робити з файлами з розширенням .ashx.

ASHX файли мають ті ж обмеження, як ASPX і ASCX файли з точки зору їх місця в проекті ASP.NET. Прості універсальні обробники йдуть з проектом. Призначені для користувача обробники можуть бути розгорнуті і розділені серед підприємств [REST], [REST2].

Створення та тестування простої служби «Hello, World» у середовищі Visual Studio .NET

Для забезпечення автоматизації процесу розробки веб-служб у середовищі Visual Studio.NET необхідно забезпечити коректну взаємодію середо-

вища з web-сервером (IIS, Apache). Для цього здійснюють конфігурацію веб-серверу із застосуванням серверних розширень FrontPage *(FrontPage Server Extensions)*. *FrontPage Server Extensions* – це технологія, яка дозволяє клієнтам Microsoft FrontPage здійснювати взаємодію з веб-сервісами та отримувати додаткову функціональність пов'язану з авторизацією, адмініструванням та користуванням веб-сайтів. Фактично, це група програм, які виконуються на сервері та ґрунтуються на застосуванні HTTP протоколу для взаємодії, та CGI/POST для серверної обробки.

Для побудови нової веб-служби використовують середовище розробки Visual Studio .NET та створюють проект за шаблоном ASP.NET Web Service, який знаходиться у розділі Visual C# Projects. Аналогічний шаблон існує в розділі проектів на мовах Visual Basic і Visual C++. Вибравши мову і шаблон, слід указати місцезнаходження файлів проекту. У нашому випадку слід задати URL-адресу веб-серверу IIS, додавши до нього ім'я папки, де розташовуватимуться файли проекту (у даному прикладі - HelloWorld). Ім'я проекту в полі Name формується автоматично і співпадає з ім'ям папки, яка містить файли проекту.

Після натискання кнопки ОК інтегроване середовище розробки (Integrated Development Environment, IDE) створює рішення (solution) і проект, а також автоматично включає в проект декілька файлів. Крім того, IDE сформує в IIS віртуальну папку з ім'ям проекту (в даному випадку - HelloWorldService).

При створенні нового проекту веб-служби ASP.NET середовище Visual Studio.NET генерує стандартний код, який містить приклад створення простої веб-служби лише з однією функцією. Приклад початкового файлу HelloWorldService.asmx.cs показано на рис.61.

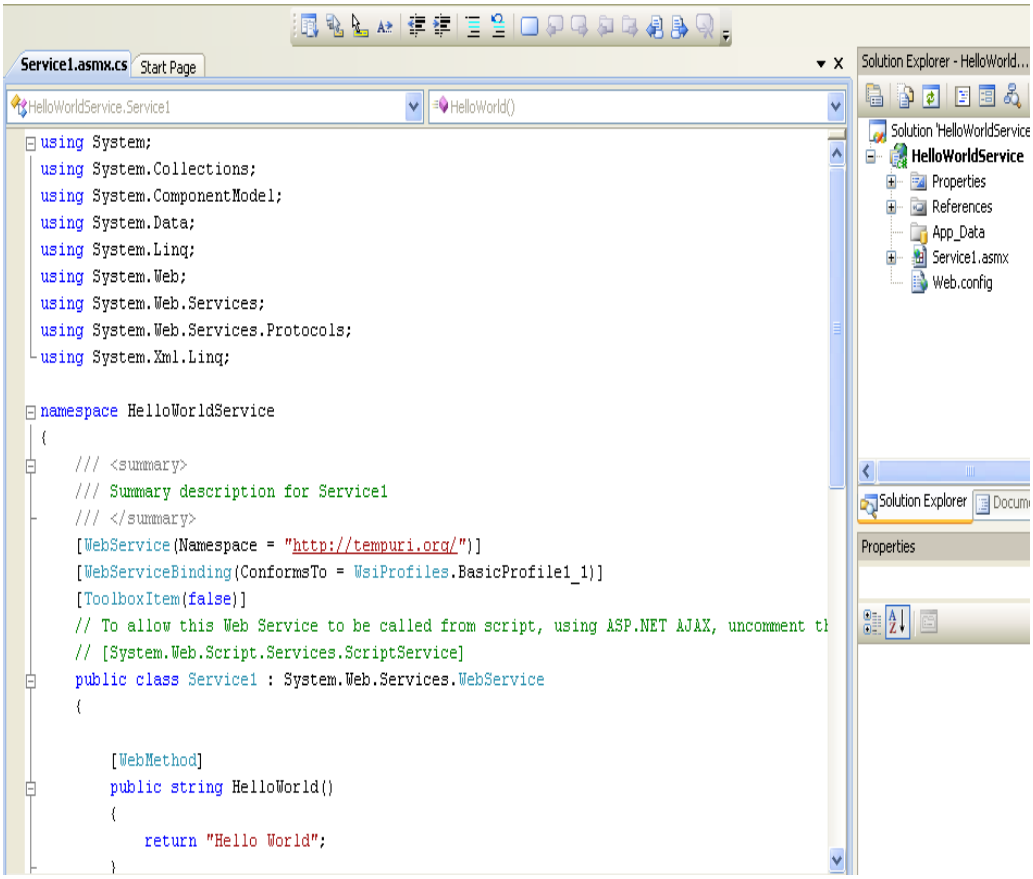



Рис.61. Стандартний код в середовищі Visual Studio .NET

Наведений стандартний код починається з операції імпортування декількох просторів імен веб-служб, які звичайно необхідні для роботи, і автоматичної генерації визначень просторів імен і класів. У даному прикладі простором імен є HelloWorldService, а класом – Service1.

Не всі простори імен, які автоматично імпортуються на початку коду, є обов'язковими. Такі простори, як System.Data, System.Collections і System.Diagnostics, за замовчанням до проекту не додаються.

На відміну від вмісту сторінок Active Server Pages, вміст сторінок веб-служб не призначено для відображення в браузері. Ці сервісні сторінки використовуватимуться клієнтськими додатками за допомогою таких протоколів, як HTTP, SMTP або SOAP через HTTP. Деякі з перелічених протоколів, зокрема SOAP, служать для взаємодії серверів, інші ж, такі як HTTP, забезпечують традиційний доступ до веб-сторінок.

Отримати доступ до веб-служб, які базуються на застосуванні транс-

портного протоколу HTTP, так само просто, як і звернутися до звичайної веб-сторінки. Для цього потрібно встановити веб-браузер на клієнтській машині. За замовчанням всі веб-служби .NET намагаються підтримувати і HTTP, і SOAP. Оскільки веб-служби є додатками, клієнтами яких також виступають додатки, і, як наслідок, вони не мають графічного інтерфейсу, призначеного для користувача, .NET надає стандартну тестову сторінку веб-служби, яка відображається при зверненні браузера до asmx-сторінки. Якщо відкрити браузер і набрати в полі URL адресу створеної веб-служби, з'явиться тестова сторінка (рис.62). Ця сторінка генерується середовищем .NET кожного разу, коли посилається запит на asmx-сторінку.

На тестовій сторінці відображається ім'я служби *HelloWorldService*, список методів (зараз лише метод *HelloWorld*()) і посилання на документ з описом служби.

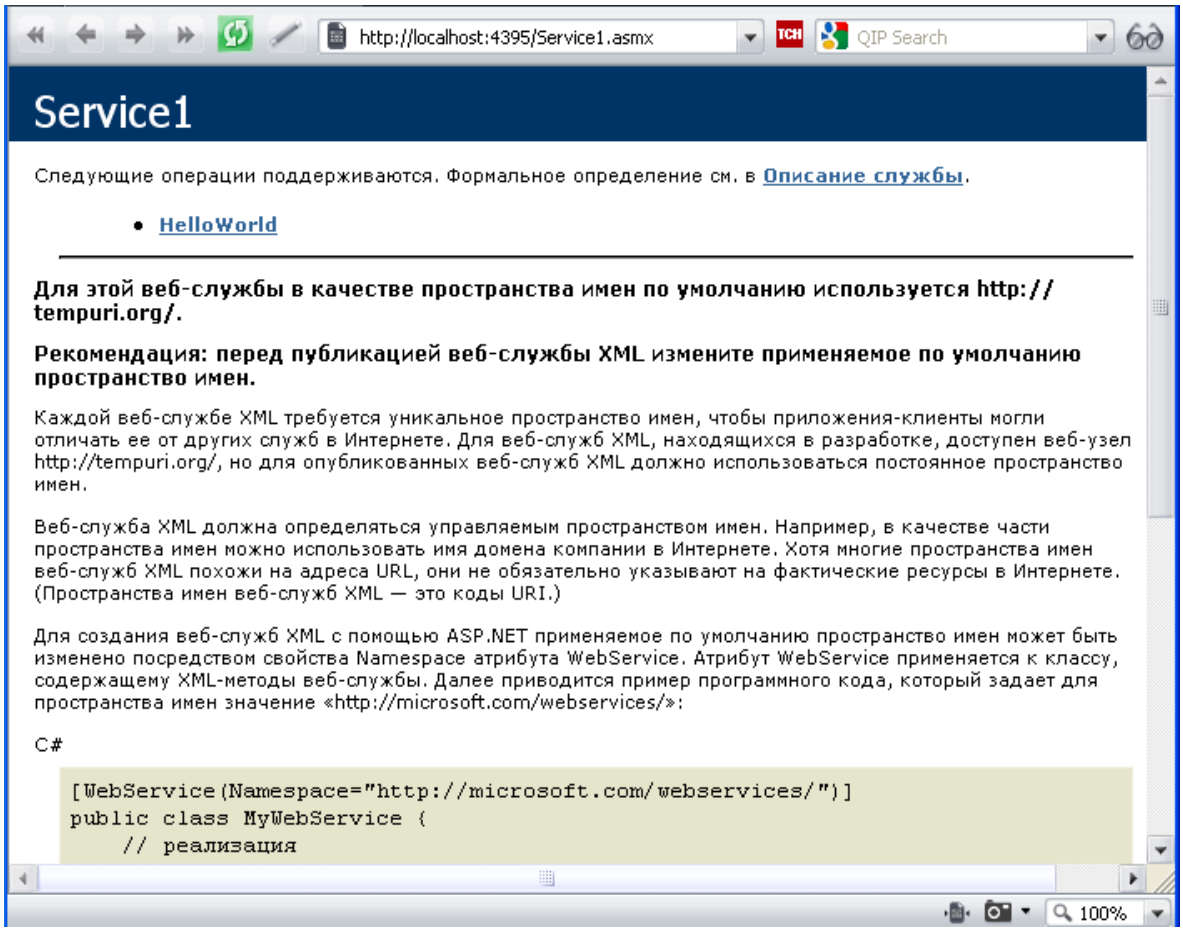



Рис.62. Тест-сторінки служби HelloWorldService

При створенні нового проекту веб-служби ASP.NET середовище Visual

Studio.NET генерує стандартний код «Hello World», який може стати стартом при розробці проекту.

Для повідомлення клієнту про можливості надані службою існують відповідні механізми документування такої інформації,- це [WebService] і [WebMethod], які є членами простору імен System.Web.Services. Для нашого проекту в атрибуті [WebMethod] встановимо властивість Description, яка описує властивості методу.

```
[WebMethod (Description="Вибирає всі рядки таблиці, які задовольняють введеним даним")]
```

## 9. Практичні приклади застосування веб-сервісів ASP.NET

### 9.1. Створення рівню логики

Для досягнення цієї мети необхідно вирішити наступні задачі.

1. Побудова БД користувачів та відповідної бізнес-логіки з функцією пошуку користувача за заданими атрибутами.
2. Побудова веб-сервісу AdministrationWebService з можливістю ідентифікації користувача, що взаємодіє з логікою БД. Публікація сервісу.
3. Створення проксі-класу. Інтеграція проксі-класу в клієнтський додаток.

#### 9.1.1. Створення БД

**Таблиця User**

| Поле | Тип | Призначення |
|---|---|---|
| ID | int | ключ |
| login | nvarchar(50) | логін |
| password | nvarchar(50) | пароль |
| usertypeid | int | Посилання на тип користувача (адміністратор, експерт, користувач) |
| state | int | Стан (має доступ в систему, не має за причиною 1, …) |
| dtbegin | datetime | Дата регістрації |
| dtend | datetime | Дата відключення від системи |

**Таблиця Usertype**

| Поле | Тип | Призначення |
|---|---|---|
| ID | int | ключ |
| description | nvarchar(50) | описання |

Існують два стандартних засоба: використання інструменту MSSQL Management Studio (рис. 63), використання вбудованого засобу Visual studio 2008 - Server Explorer (додаємо з'єднання, вказуємо ім'я серверу, ім'я БД) (рис.64).

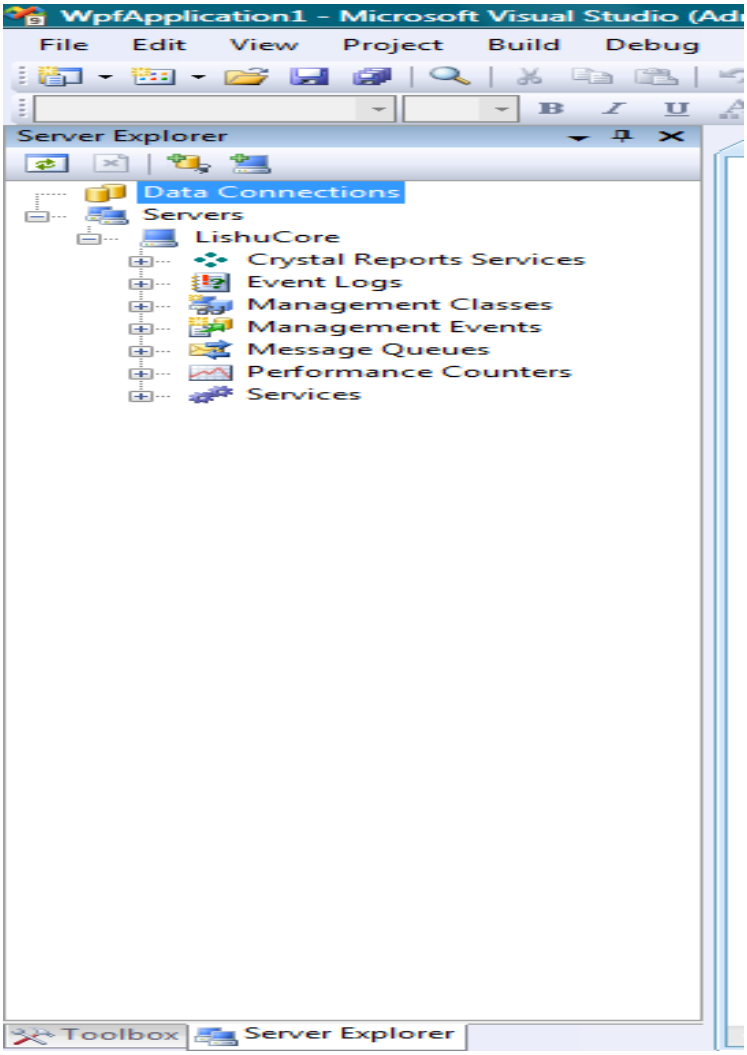



Рис.63. Інструмент MSSQL Management Studio

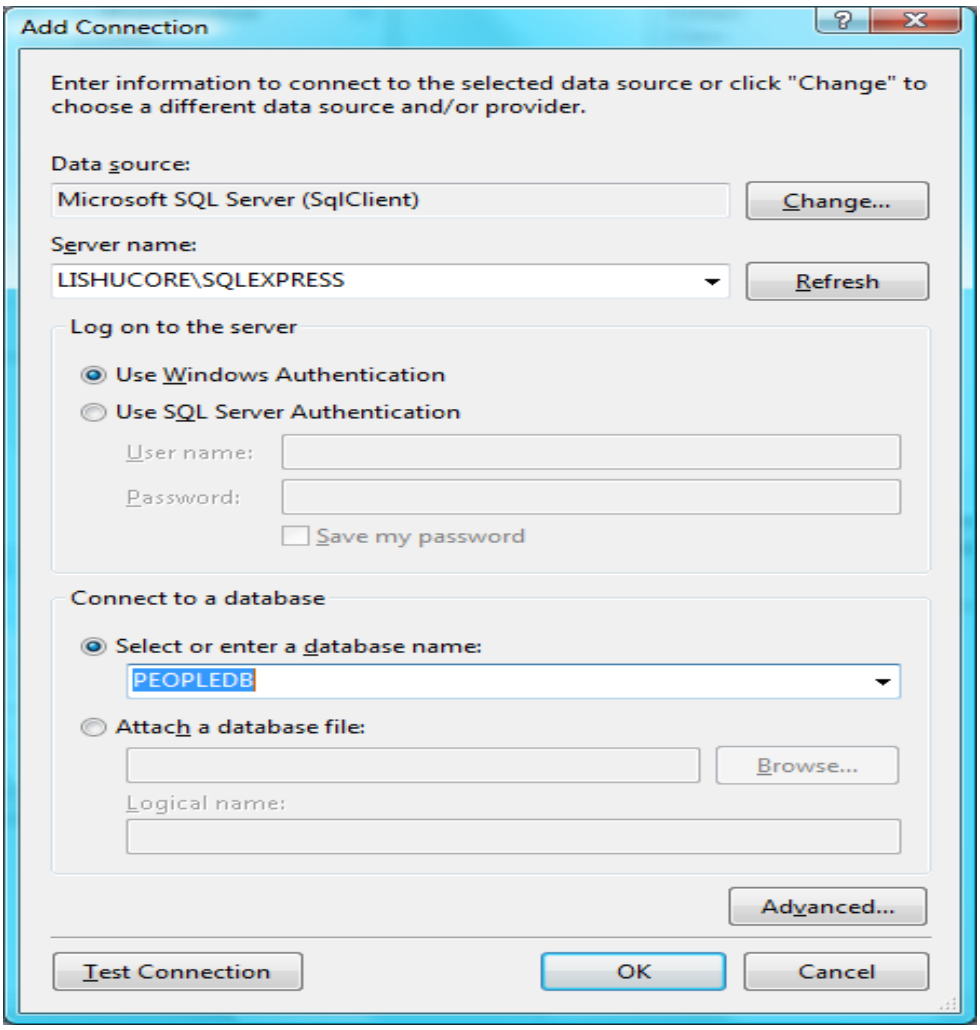



Рис.64. Вбудований засіб Visual studio 2008 - Server Explorer

Створюємо процедуру, що повертає користувача: виділяємо опцію *Stored Procedures* в дереві БД, викликаємо контекстне меню, обираємо *Add New Stored Procedure*. В полі вводимо команду:

```
create PROCEDURE GetUser
@login nvarchar(50),
@password nvarchar(50)
AS
SELECT * FROM [user] WHERE [login] = @login AND [password] = @password
```

Зберігаємо: правою кнопкою миші викликаємо меню Save.

Здійснюємо тестування – вводими в окно код

```
use xadministration
go
GetUser "admin", "admin"
Go
```

В таблицю user включаємо користувача . Запускаємо на виконання процедуру. Одержуємо заданого користувача.

### 9.1.2.Створення рівню DAL

Створюємо рішеня (solution) з ім’ям Administration. Додаємо новий проект Administration.DAL типу бібліотека.

Додаємо клас UserDAL.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Linq;
using System.Text;
using System.Configuration;
namespace Administration.DAL
{
   public class UserDAL
   {
      private string connectionString;
      SqlConnection sqlConnection;
      public UserDAL()
      {
         try
         {
            connectionString = ConfigurationManager.AppSettings["ConnectionString"];
            sqlConnection = new SqlConnection(connectionString);
```

```
    }
    catch (Exception ex)
    {
        throw new Exception(ex.Message, ex);
    }
}
public void CloseConnection()
{
    try
    {
        sqlConnection.Close();
    }
    catch (Exception ex)
    {
        throw new Exception(ex.Message, ex);
    }
}
public IDataReader GetUser(string login, string password)
{
    try
    {
        SqlCommand cmd = new SqlCommand("GetUser", sqlConnection);
        cmd.CommandType = CommandType.StoredProcedure;
        cmd.Parameters.AddWithValue("@login", login);
        cmd.Parameters.AddWithValue("@password", password);
        try
        {
            if(sqlConnection.State != ConnectionState.Open)
            sqlConnection.Open();
            SqlDataReader reader = cmd.ExecuteReader();
            return (IDataReader) reader;
        }
        catch (Exception ex)
        {
            throw new Exception(ex.Message, ex);
```

```
            }
        }
        catch (Exception ex)
        {
            throw new Exception(ex.Message , ex);
        }
    }
  }
}
```

Для тестування логіки створюємо проект Administration.DAL.Test типу консольний додаток. Додаємо посилання на зборку Administration.DAL. Додаємо новий клас DALTester з методом GetUserTest().

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Text;
namespace Administration.DAL.Test
{
  public class DALTester
  {
    public void GetUserTest()
   {
     try
      {
       UserDAL userDAL = new UserDAL();
       IDataReader dataReader = userDAL.GetUser("admin", "admin");
         while (dataReader.Read())
          {
           Console.WriteLine(dataReader["active"].ToString());
           // Add to collection

          }
          dataReader.Close();
```

```
 //dataReader.
            }
            catch(Exception ex)
            {
              Console.WriteLine(ex.Message + "\n" + ex.StackTrace);
            }
        }
    }
}
```

Додаємо клас TestRunner

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Administration.DAL.Test
{
    public class TestRunner
    {
        static void Main()
        {
            DALTester dalTester = new DALTester();
            dalTester.GetUserTest();
            Console.ReadKey();
        }
    }
}
```

Додаємо в секцію appSettings файлу конфігурації App.config строку підключення до БД:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
        <appSettings>
```

```
        <add key="connectionString" value="Server = LISHUCORE;Database =
xadministration; UID = sa; PWD = sa;Connection Lifetime = 0; Max Pool Size = 300;Min Pool
Size = 10; Pooling = true;" />
    </appSettings>
</configuration>
```

Запускаємо проект на виконання.

### 9.1.3. Створення рівню BLL

Рівень бізнес логіки відповідає за автоматизацію бізнес процесів. Він розділяється на дві частини: структурну (бізнес об’єкти ) та операційну (функції). В нашому випадку бізнес об’єктами є: користувач та список користувачів; функцією: ідентифікація користувача в системі.

Рівень BLL пов'язаний з рівнем DAL. Тому першим кроком ставимо посилання на збірку Administration.DAL.

Далі додаємо бізнес об’єкти.

Користувач

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Administration.BLL
{
   [Serializable]
   public class User
   {
      public int ID { get; set; }
      public int Active { get; set; }
      public string Login { get; set; }
      public string Password { get; set; }
      public User()
      {

      }
      public User(int _id, string _login, string _password, int _active)
```

```
        {
            this.ID = _id;
            this.Login = _login;
            this.Password = _password;
            this.Active = _active ;
        }
    }
}
```

Колекція користувачів

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Administration.BLL
{
    [Serializable]
    public class UserCollection:List<User>
    {
    }}
```

Важливою особливістю завдання бізнес класів є використання атрибуту Serializable – це дозволяє виконати серіалізацію/десеріалізацію об’єкта класу для здійснення обміну за стандартним протоколом.

Клас, що відповідає за функціональну частину, називається менеджером користувачів.

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Text;
using Administration.DAL;
namespace Administration.BLL
{
```

```
public class UserManager
{
   public User GetUser(String login, String password)
   {
      try
      {
         UserCollection userCollection = new UserCollection();
         UserDAL userDAL = new UserDAL();
         IDataReader dataReader = userDAL.GetUser(login, password);
         while (dataReader.Read())
         {
            User user = new User((int) dataReader["ID"],
                                 (string) dataReader["Login"],
                                 (string) dataReader["Password"],
                                 (int) dataReader["Active"]);
            userCollection.Add(user);
         }
         if (userCollection.Count > 1)
            throw new Exception(
"Get user exception: each user should be identified by the login and password rule violation");
         else
            return userCollection[0];
      }
      catch(Exception ex)
      {
         throw new Exception(ex.Message, ex);
      }
   }
}
```

Створюємо тестер

Для тестування логіки створюємо проект Administration.BLL. Test типу консольний додаток. Додаємо посилання на зборки Administration.BLL, Administration.DAL.

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Text;
using Administration.BLL;
namespace Administration.BLL.Test
{
   class BLLTester
   {
      public void GetUserTest()
      {
         try
         {
            UserManager userManager = new UserManager();
            User user = userManager.GetUser("admin", "admin");
            Console.WriteLine("ID: "+ user.ID.ToString() +"\n"+
                        "Login: "+ user.Login.ToString() +"\n"+
                        "Password: "+ user.Password.ToString() +"\n"+
                        "Active: "+ user.Active.ToString()
               );
         }
         catch (Exception ex)
         {
            Console.WriteLine(ex.Message + "\n" + ex.StackTrace);
         }
      }
   }
}
```

Додаємо клас TestRunner.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Administration.BLL.Test
{
   class TestRunner
   {
      static void Main(string[] args)
      {
         BLLTester bllTester = new BLLTester();
         bllTester.GetUserTest();
         Console.ReadKey();
      }
   }
}
```

Додаємо в секцію appSettings файлу конфігурації App.config строку підключення до БД:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
        <appSettings>
                <add key="connectionString" value="Server = LISHUCORE;Database = xadministration; UID = sa; PWD = sa;Connection Lifetime = 0; Max Pool Size = 300;Min Pool Size = 10; Pooling = true;" />
        </appSettings>
</configuration>
```

Запускаємо тест.

### 9.2. Створення рівню сервіса

Створюємо веб-сервіс AdministrationWebService. Додаємо новий проект типу ASP.NET Web service (рис.65).

Створюємо solution з проектом. Зв'язуємо сервіс із віртуальною папкою IIS.

Результат тесту: http://localhost/AdministrationWebService/Service1.asmx

Додаємо посилання на збірки Administration.BLL, Administration.DAL

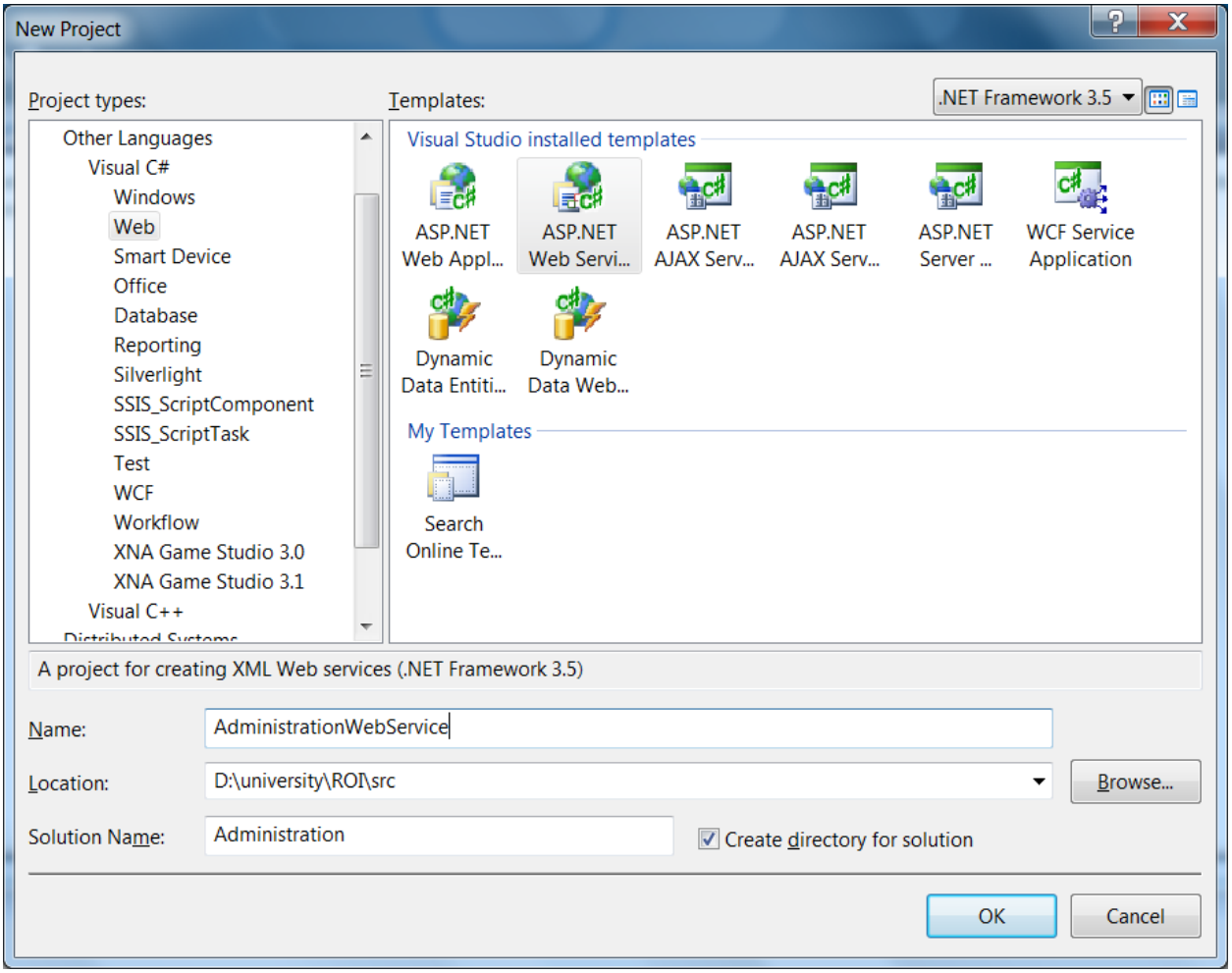



Рис.65. Створення рівню сервіса

Перейменуємо сервіс у AdministrationService, додаємо метод отримання користувача:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Services;
using Administration.BLL;
```

```
namespace AdministrationWebService
{
   [WebService(Namespace = "http://tempuri.org/")]
   [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
   public class AdministrationService : System.Web.Services.WebService
   {
      [WebMethod]
      public User GetUser(String login, String password)
      {
         try
         {
            UserManager userManager = new UserManager();
            User user = userManager.GetUser(login, password);
            return user;
         }
         catch (Exception ex)
         {
            throw new Exception(ex.Message, ex);
         }
      }
   }
}
```

В результаті одержимо наступне (рис.66 ).

В файл Web.config додаємо строку

```
<appSettings>
                <add key="connectionString" value="Server = LISHUCORE;Database = xadministration; UID = sa; PWD = sa;Connection Lifetime = 0; Max Pool Size = 300;Min Pool Size = 10; Pooling = true;" />
            </appSettings>
```

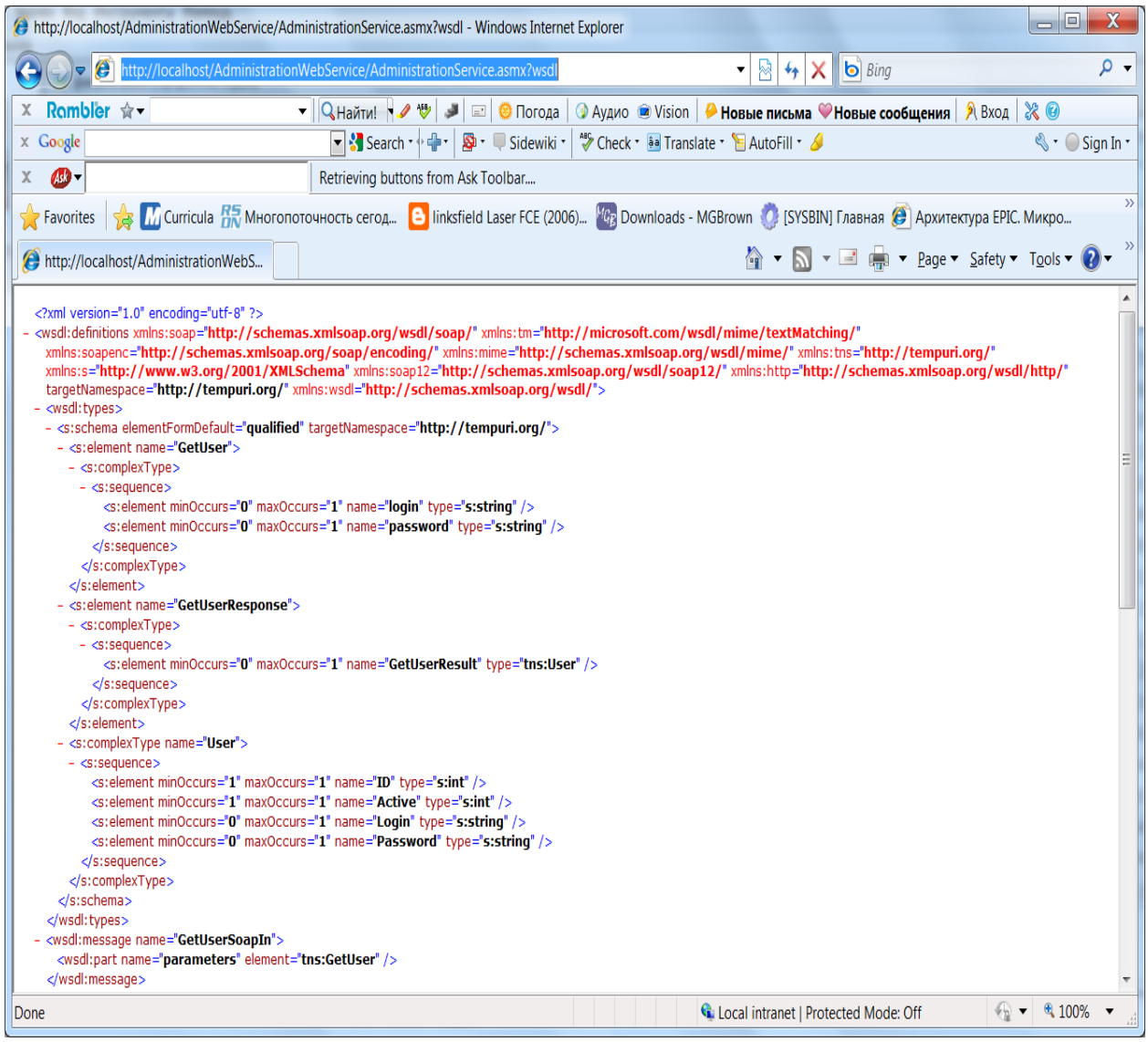



Рис.66. Вид контракту сервісу

### 9.3. Створення тестеру клієнта

Для тестування логіки створюємо проект Administration.WebService. Test типу консольний додаток. Тестування проводимо на двох рівнях – грубому (з використанням HTTP запитів) та звичайному.

Грубий метод. Створюємо клас RawWebServiceTest. Метод TestSoap формує HTTP повідомлення з включеним в нього повідомленням SOAP, яке асинхронно відсилається з використанням методу *BeginGetResponse*.

```
using System;
using System.Collections.Generic;
using System.IO;
```

```
using System.Linq;
using System.Net;
using System.Text;
using System.Xml;
namespace Administration.WebService.Test
{
   public class RawWebServiceTester
   {
      string _soapEnvelope =
            @"<soap:Envelope
               xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
               xmlns:xsd='http://www.w3.org/2001/XMLSchema'
               xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'>
            <soap:Body></soap:Body></soap:Envelope>";
      private  XmlDocument CreateSoapEnvelope(string content)
      {
         StringBuilder sb = new StringBuilder(_soapEnvelope);
         sb.Insert(sb.ToString().IndexOf("</soap:Body>"), content);
         // create an empty soap envelope
         XmlDocument soapEnvelopeXml = new XmlDocument();
         soapEnvelopeXml.LoadXml(sb.ToString());
         return soapEnvelopeXml;
      }
      private  HttpWebRequest CreateWebRequest(string url, string action)
      {
         HttpWebRequest webRequest = (HttpWebRequest)WebRequest.Create(url);
         webRequest.Headers.Add("SOAPAction", action);
         webRequest.ContentType = "text/xml;charset=\"utf-8\"";
         webRequest.Accept = "text/xml";
         webRequest.Method = "POST";
         return webRequest;
      }

      private  void InsertSoapEnvelopeIntoWebRequest(XmlDocument
soapEnvelopeXml, HttpWebRequest webRequest)
```

```
            {
                using (Stream stream = webRequest.GetRequestStream())
                {
                    soapEnvelopeXml.Save(stream);
                }
            }
            public void TestSoap(String _url, String _action)
            {
                string _outputPath = @"C:\1\output.xml";
                string content = @"<GetUser xmlns=""http://tempuri.org/"">
                                    <login >admin</login>
                                    <password >admin</password>
                                   </GetUser>";
                XmlDocument soapEnvelopeXml = CreateSoapEnvelope(content);
                HttpWebRequest webRequest = CreateWebRequest(_url, _action);
                InsertSoapEnvelopeIntoWebRequest(soapEnvelopeXml, webRequest);
                // begin async call to web request.
                IAsyncResult asyncResult = webRequest.BeginGetResponse(null, null);
                // suspend this thread until call is complete. You might want to
                // do something usefull here like update your UI.
                asyncResult.AsyncWaitHandle.WaitOne();
                // get the response from the completed web request.
                string soapResult;
                using (WebResponse webResponse =
webRequest.EndGetResponse(asyncResult))
                using (StreamReader rd = new
StreamReader(webResponse.GetResponseStream()))
                {
                    soapResult = rd.ReadToEnd();
                }
                File.WriteAllText(_outputPath, soapResult);
            }

        }
    }
```

Створюємо TestRunner.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Administration.WebService.Test
{
   class TestRunner
   {
      static void Main(string[] args)
      {
         RawWebServiceTester rawWebServiceTester = new RawWebServiceTester();
         WebServiceTester webServiceTester = new WebServiceTester();
rawWebServiceTester.TestSoap("http://localhost/AdministrationWebService/AdministrationServ
ice.asmx", "http://tempuri.org/GetUser");
         Console.ReadKey();
      }
   }
}
```

В результаті отримуємо файл C:\1\output.xml.

```
<?xml version="1.0" encoding="utf-8" ?>
- <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
- <soap:Body>
   - <GetUserResponse xmlns="http://tempuri.org/">
      - <GetUserResult>
         <ID>1</ID>
         <Active>1</Active>
         <Login>admin</Login>
         <Password>admin</Password>
      </GetUserResult>
```

```
    </GetUserResponse>
</soap:Body>
</soap:Envelope>
```

Звичайний метод побудови клієнту складається в наступному. В проекті додаємо посилання на веб-сервіс (рис.67).

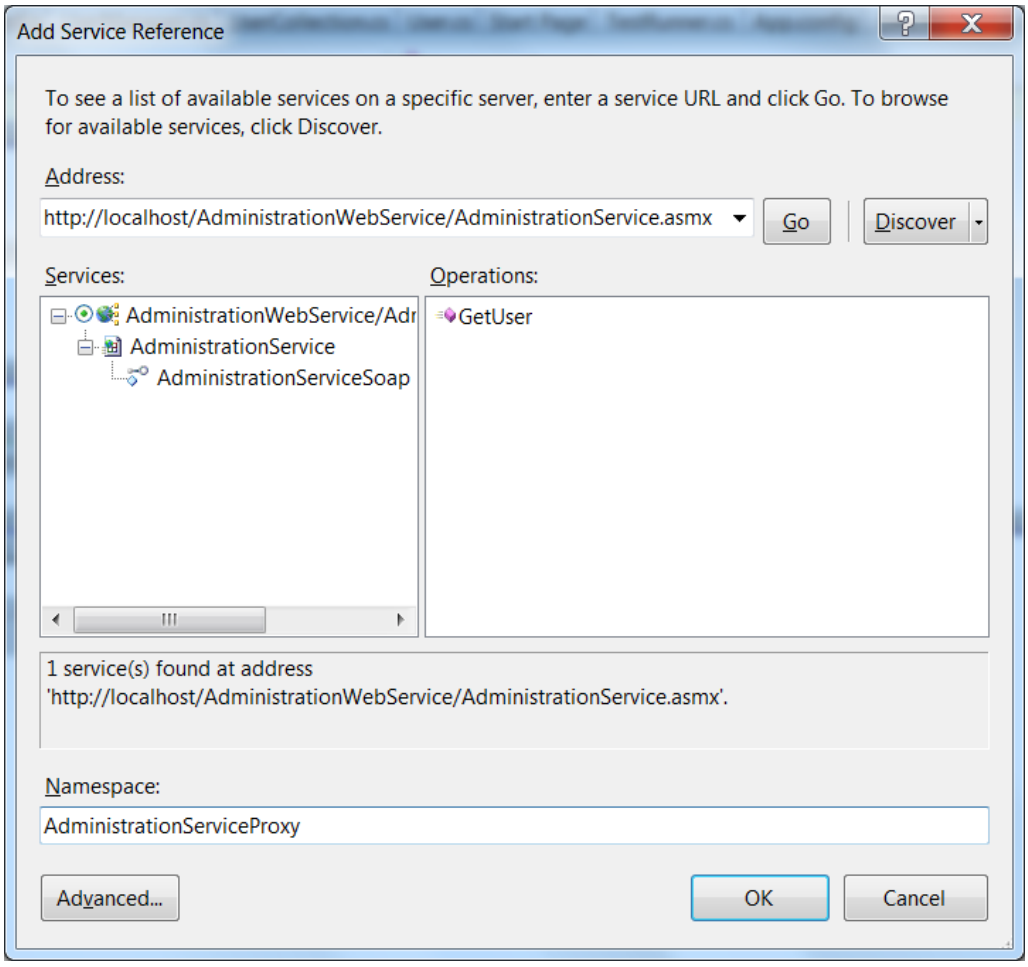



Рис.67. Посилання на веб-сервіс

Створюємо відповідний клас.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Administration.WebService.Test.AdministrationServiceProxy;

namespace Administration.WebService.Test
```

```
{
    public class WebServiceTester
    {
        public void GetUser()
        {
            //Administration
            AdministrationServiceSoapClient administrationServiceSoapClient =
                                        new AdministrationServiceSoapClient();
         User user = administrationServiceSoapClient.GetUser("admin", "admin" );
         Console.WriteLine("ID: " + user.ID.ToString() + "\n" +
                        "Login: " + user.Login.ToString() + "\n" +
                        "Password: " + user.Password.ToString() + "\n" +
                        "Active: " + user.Active.ToString()
                    );
        }
    }
}
```

Змінюємо TestRunner.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Administration.WebService.Test
{
    class TestRunner
    {
        static void Main(string[] args)
        {
            RawWebServiceTester rawWebServiceTester = new RawWebServiceTester();
            WebServiceTester webServiceTester = new WebServiceTester();


//rawWebServiceTester.TestSoap("http://localhost/AdministrationWebService/AdministrationSe
rvice.asmx",
                //    "http://tempuri.org/GetUser");
```

```
            webServiceTester.GetUser();
            Console.ReadKey();
        }
    }
}
```

Запускаємо тест (рис.68) .

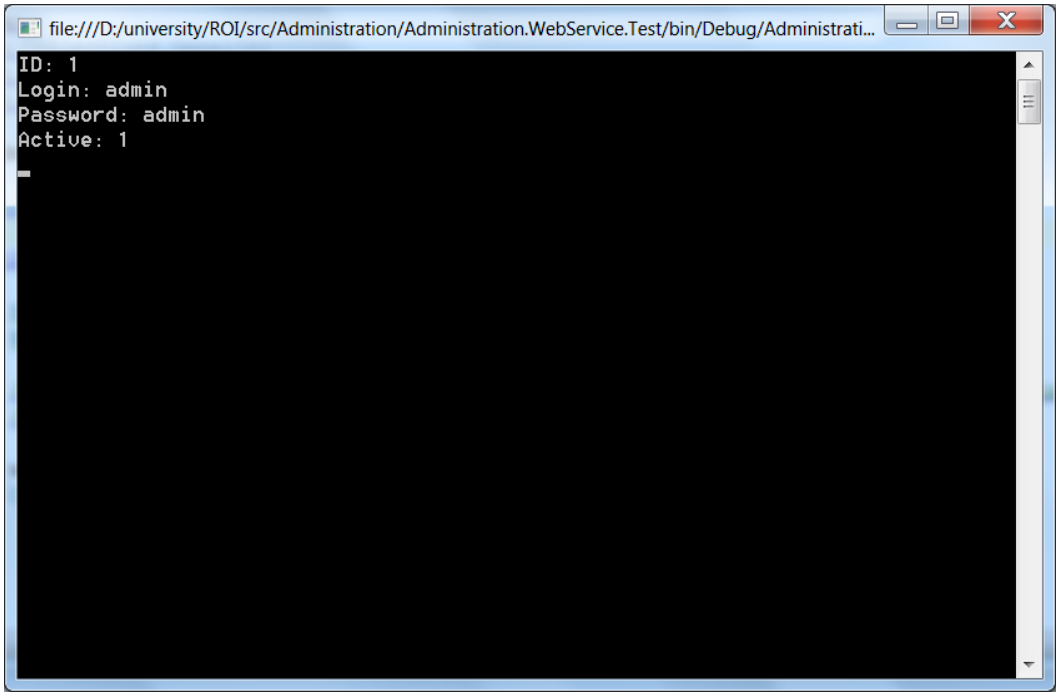



Рис.68. Результати тестування

### 9.4. Завдання для самостійної підготовки

1. Реалізувати метод, який повертає колекцію користувачів, методи додавання та видалення користувача.
2. Реалізувати кешування довідників (наприклад, вулиць, стран, регіонів).
3. Дослідити властивості веб-сервісу, пов’язані з включенням режиму сесії, кешування. Провести стрес тестування, дослідити затримки обробки запитів.
4. Реалізувати зв'язок сервісів при реалізації задачі отримання всіх даних по пацієнту лікувального закладу в розподіленій системі. Підсистеми лабораторія, стаціонар, поліклініка існують відокремлено .
5. Повернути список пацієнтів за завданими умовами. Наприклад, всіх пацієнтів з відповідним діагнозом.

# 10. Практична робота з сокетами засобами C#

## 10.1. Визначення та описання сокетів

Міжкомп'ютерна взаємодія, що складає основу будь якої розподіленої обробки інформації, базується на сьомірівневої еталонної моделі взаємодії відкритих систем **(**open system interconnection - OSI).

Суть моделі складається в застосуванні багаторівневої архітектури, де кожний об'єкт, що знаходиться на рівні n, взаємодіє тільки з віддаленими об'єктами n-го рівня, при цьому об'єкти n-го рівня використовують служби, які забезпечуються об'єктами рівня (n-1). Об'єкти на рівні n обмінюються протокольними одиницями обміну (protocol data units - PDU), які містять у собі заголовок n-го рівня та данні n-го рівня, при цьому PDU n-го рівня стають даними для рівня n-1.

Основна найбільш розповсюджена реалізація даної моделі – *Internet Protocol Suite*, або «стек протоколів *TCP/IP*» (*Transmission Control Protocol/Internet Protocol*) (набір мережевих протоколів різних рівнів для організації глобальних мереж). Слід відзначити, що це найбільш розповсюджена, але не єдина реалізація.

В моделі OSI стек TCP/IP реалізує усі рівні та сам розподіляється на 4 рівня: прикладний, транспортний, міжмережевий, рівень доступу к мережі. На стеці протоколів TCP/IP побудована вся взаємодія в мережі, від програмного додатку до канального рівня моделі OSI. При цьому стек є незалежним від фізичного середовища передачі даних.

Хоча модель TCP/IP не співпадає з еталоном OSI існує наступне порівняння: Internet Application Layer виконує функції Application Layer, Prese?tation Layer, та більшу частину функцій Session Layer; Transport Layer виконує функції Session та Transport Layer; Internet Layer виконує частину функцій Network Layer; Link Layer виконує функції Data Link, Physical Layers, та частину функцій Network Layer.

Структура стеку протоколів показана на рис.69.

<table>
<tr><td>HTTP<br>Web-protocol</td><td>FTP<br>File trans-fer protocol</td><td>SMTP<br>Simple mail transfer pro-tocol</td><td>NFS<br>Network file system<br>RPC<br>Remote pro-cedure call</td><td>DNS<br>Domain name server</td><td>SNMP<br>Simple net-work man-agement protocol</td></tr>
<tr><td colspan="3">TCP</td><td colspan="3">UDP</td></tr>
<tr><td colspan="6">IP</td></tr>
<tr><td colspan="6">Рівень каналу передачі даних (Ethernet, WiFi…)</td></tr>
</table>

Рис.69. Стек протоколів

Великий внесок у розвиток стеку протоколів TCP/IP вніс університет Берклі, реалізувавши протоколи стека у своїй версії ОС UNIX. З реалізацією університетом Берклі пов’язане поняття сокетів, яке стало основною абстракцією програмування мережевої взаємодії.

Термін «сокет» означає кінцеву точку двостороннього каналу зв'язку між двома програмами за протоколом TCP/IP, яка визначається характеристиками віртуального каналу: протокол, локальний IP адрес та порт, віддалений адрес та порт. Адрес сокету складається з адреси комп’ютера та порту, який відображається операційною системою на додаток (process) або нитку (thread), що приймає учать у взаємодії.

Термін «сокети» (Internet sockets) означає механізм передачі пакетів до відповідного процесу або нитки, та відповідний програмний інтерфейс (API) для забезпечення функціональності цієї міжпроцесної, як правило мережевої, взаємодії.

Поняття сокету в прикладних додатках — це покажчик на структуру даних (в C# це тип), поля якої задають параметри віртуального канала. Такими параметрами є: сім’я адрес, тип комунікації та протокол.

Сім’я адрес (address family) характеризує набір протоколів, що відповідає даному сокету (InterNetwork, Unix, Appletalk, Banyan, Atm, Ipx та інше).

Тип комунікації визначає метод міжпроцесної взаємодії: зі встановленням з'єднання (stream); без з'єднання (datagram); рівня транспорту TCP/UDP (raw); і характеризує роботу додатка на мережевому рівні (IP).

Життєвий цикл сокету складається з: формування, відкриття, обміну, закриття. Код створення сокету з підтримкою TCP/IP засобами C# представляється наступним чином

```
using System.Net.Sockets;
…
Socket slistener = new Socket(AddressFamily.InterNetwork,
        SocketType.Stream,
        ProtocolType.Tcp);
```

Для забезпечення підтримки стеку Unix

```
Socket slistener = new Socket(AddressFamily.Unix,
                SocketType.Stream,
                ProtocolType.IP).
```

Параметр STREAM служить для побудови надійного двонаправленого каналу обміну, орієнтованого на з'єднання (TCP для InterNetwork).

Далі сокет, як правило, зв'язується з конкретною адресою (кінцевою точкою доступу), що складається з IP-адреси та номеру порта. В C# це робиться з застосуванням типу IPEndPoint та методу Bind обєкту типу Socket. Якщо значення номера порту завдати рівним нулю, слоєм windows socket буде присвоєне вільне значення в диапазоні 1024-5000.

Далі можливі два варіанти поведінки сокету: ініціювання з'єднання (активна) або слухання порту (пасивна).

Нижче наведений приклад відкриття та ініціалізації пасивного сокету (сокет-сервера).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
```

```
using System.Net.Sockets;
using System.Text;
namespace SocketTest
{
   class Program
   {
      static void Main(string[] args)
      {
         IPEndPoint localIP = new IPEndPoint(IPAddress.Any, 11000);
         Socket slistener = new Socket(AddressFamily.InterNetwork,
                                       SocketType.Stream,
                                       ProtocolType.Tcp);
         slistener.Bind(ipEndPoint);
         slistener.Listen(10);
      }
   }
}
```

У разі активного сокету (сокет клієнту) в метод Connect здійснюється передача адреси додатку сервера.

```
IPHostEntry ipTargetHost = Dns.GetHostEntry("192.168.58.176");
IPAddress ipTargetAddr = ipTargetHost.AddressList[0];
IPEndPoint ipTargetEndPoint = new IPEndPoint(ipTargetAddr, 11000);
   senderSock = new Socket(AddressFamily.InterNetwork,
                                   SocketType.Stream,
                           ProtocolType.Tcp);
senderSock.Connect(ipTargetEndPoint).
```

Якщо з'єднання встановлене, то починається взаємодія з використанням методів Send та Receive. Як правило ця взаємодія організується на базі легковагомого фонового процесу (thread) :

```
if (senderSock.Connected == true)
```

```
{
        tr = new Thread(socketRec);
        tr.IsBackground = true;
        tr.Start();
        break;
 }
```

### 10.1.1. Типи комунікації сокетів. Потокова комунікація

Як, вже було вказано, існують три основні типи комунікації з застосуванням механізму сокетів: потік, дейтаграма, низький рівень. В основі потокової комунікації лежить протокол Transmission Control Protocol (TCP), який забезпечує потоковий інтерфейс, тобто приймає від додатку потік байтів, які необхідно передати через відкритий канал-з'єднання. При цьому протокол гарантує, що дані будуть передані в потрібній послідовності і без помилок. Слід відзначити можливість двонаправленого зв'язку. Такий спосіб комунікації доцільно використовувати для передачі великих об'ємів даних, оскільки накладні витрати, пов'язані зі встановленням окремого з'єднання для кожного повідомлення, що відправляється, може виявитися неприйнятним для невеликих об'ємів даних. Протокольна одиниця обміну на даному рівні – сегмент. Заголовок сегменту складає 20 байтів. Протокол TCP посилає сегменти в рамках IP-дейтаграм.

Особливістю протоколу є встановлення з'єднання, тобто формування шляху до початку передачі повідомлень. Тим самим гарантується, що обидві сторони, що беруть участь у взаємодії, приймають і відповідають. Якщо додаток відправляє одержувачу, два повідомлення, то гарантується, що ці повідомлення будуть одержані в тій же послідовності. Проте окремі повідомлення можуть дробитися на пакети. При використовуванні TCP цей протокол бере на себе розбиття передаваємих даних на пакети відповідного розміру, відправку їх в мережу і збірку їх на іншій стороні. Додаток знає тільки, що він відправляє на рівень TCP певне число байтів і інша сторона одержує ці бай-

ти. У свою чергу TCP ефективно розбиває ці дані на пакети відповідного розміру, відправляє їх в мережу, одержує ці пакети на іншій стороні, виділяє з них дані і об'єднує їх разом.

Для здійснення з'єднання використовується механізм, що має назву трьохетапне квітирування зв'язку (three-way hadshake).

За цим механізмом, хост А посилає сегмент синхронізації (SYN-сегмент) хосту В, хост В у відповідь посилає сегмент синхронізації хосту А, який у відповідь посилає сегмент підтвердження(АСК-сегмент). Після чого процес з'єднання вважається закінченим. Час від посилки сегменту синхронізації до прийому відповіді складає один RTT (round trip time).

Закриття з'єднання називається полу-закриттям (half-close) тому, що коли хост закриває з'єднання, він указує цим, що більш не буде посилати дані іншому хосту, але може приймати дані. Повне закриття з'єднання здійснюється, коли обидва хости ініціюють закриття: хост А посилає сегмент FIN хосту В, хост В посилає АСК-сегмент та сегмент FIN хосту А і отримує від нього АСК-сегмент.

Таким чином для установлення з'єднання необхідне здійснення передачі 3-х сегментів і для завершення 4-х.

Для контролю помилок протокол використовує АСК-сегменти, таймаути та повторні передачі. Керування потоком здійснюється з використанням механізма ковзного вікна (sliding window). Більш детальна інформація наведена в [ http://www.rhyshaden.com/tcp.htm].

### 10.1.2. Дейтаграмний тип комунікації

Цей тип базується на протоколі User Datagram Protocol (UDP), який надає можливість надсилати повідомлення з мінімальними затратами есурсів і часу (minimum of protocol mechanism). Протокол орієнтований на транзакції, але надійна доставка даних та захист від дублювання не гарантовані (is not very reliable (but fast) connectionless protocol).

Між сокетами у цьому разі не встановлюється ніякого явного з'єднання – повідомлення відправляється вказаному сокету і, відповідно, може виходити від вказаного сокета.

Потоковий тип взаємодії в порівнянні з дейтаграмним дійсно надає надійніший метод, але для деяких додатків накладні витрати, пов'язані з установкою явного з'єднання, неприйнятні (наприклад, сервер часу доби, що забезпечує синхронізацію часу для своїх клієнтів). На встановлення надійного з'єднання з сервером потрібен час, який вносить затримки в обслуговування. Для скорочення накладних витрат використовуються дейтаграмні сокети [http://www.faqs.org/rfcs/rfc768.html].

Сирі сокети. Окрім двох розглянутих типів існує також узагальнена форма сокетів, яку називають необроблюваними, сирими сокетами (raw sockets). Головна мета використовування сирих сокетів полягає в обході механізму, за допомогою якого операційна система обробляє дані TCP/IP. Це досягається забезпеченням спеціальної реалізації стека TCP/IP – пакет безпосередньо передається додатку і обробляється ефективніше, ніж при проходженні через головний стек протоколів клієнта.

За визначенням сирий сокет – це сокет, який приймає пакети, обходить рівні TCP і UDP в стеку TCP/IP і відправляє їх безпосередньо додатку. При використанні таких сокетів пакет не проходить через фільтр TCP/IP, тобто ніяк не обробляється, і предстає в своїй сирій формі. У такому разі відповідальність за правильну обробку повідомлень: розбір, аналіз заголовків, формування пакетів, лягає на програміста. Це теж саме, що включити в додаток два верхніх рівня стеку TCP/IP. Сирі сокети головним чином використовуються при розробці спеціалізованих низькорівневих протокольних додатків. Наприклад, такі утиліти TCP/IP, як traceroute, ping або arp, використовують сирі сокети.

Робота з сирими сокетами вимагає солідного знання базових протоколів TCP/UDP/IP. В даній роботі вони розглядатися не будуть – детальна ін-

формація може бути знайдена на сайті [http://codeidol.com/csharp/csharp-network/ICMP/Using-Raw-Sockets/].

### 10.1.3. Порти TCP/UDP

Порт – 2 байтна константа , що служить для ідентифікації додатку на комп'ютері. З його допомогою розширяється поняття IP-адреси. Комп'ютер, на якому одночасно виконується декілька додатків, може ідентифікувати цільовий процес, користуючись унікальним номером порту.

Адреса сокету, як вже було вказано, складається з IP-адреси машини і номера порту, використовуваного додатку TCP. Оскільки IP-адреса унікальна в Internet, а номери портів унікальні, по міньшій мірі, на окремій машині, номери сокетів також унікальні у всій мережі Internet. Ця характеристика дозволяє процесу спілкуватися через мережу з іншим процесом виключно на підставі номеру сокета.

За певними службами (протоколами) номери портів зарезервовані - це широко відомі номера портів, наприклад порт 20 використовується для даних FTP або 80 - для HTTP. Додаток також може користуватися будь-яким номером порту, який не був зарезервований і поки не зайнятий. Агентство Internet Assigned Numbers Authority (IANA) веде перелік широко відомих номерів портів (http://www.iana.org/assignments/port-numbers).

Звичний додаток клієнт-сервер, використовуючи сокети, складається з двох різних додатків: клієнта, що ініціює з'єднання з сервером і серверу, чекаючого з'єднання від клієнта.

## 10.2. Робота з сокетами в .NET

Для програмування взаємодії процесів на базі сокетів в операційній системі Windows використовується шар Windows WinSock API.

.NET Framework пропонує додатковий рівень абстракції - системи класів, які забезпечують взаємодію з базовим рівнем Winsock network API. Підтримку сокетів в .NET забезпечують класи в просторі імен System.Net.Sockets.

*Socket.* Забезпечує базову функціональність додатку сокета.

*TcpClient.* Клас TcpClient будується на класі Socket, з метою забезпечення TCP-обслуговування на вищому рівні. TcpClient надає декілька методів для відправки і отримання даних через мережу.

*TcpListener.* Цей клас також побудований на низькорівневому класі Socket. Його основне призначення - серверні додатки. Він чекає вхідні запити на з'єднання від клієнтів і повідомляє додаток про будь-які з'єднання.

*UdpClient.* UDP - це протокол, не організуючий з'єднання, отже, для реалізації UDP-обслуговування в .NET потрібна інша функціональність. Клас UdpClient призначений для реалізації UDP-обслуговування.

*MulticastOption*. Клас MulticastOption встановлює значення IP-адреси для приєднання до IP-групи або для виходу з неї.

*NetworkStream*. Клас NetworkStream реалізує базовий клас потоку, з якого дані відправляються і в який вони входять. Це абстракція високого рівня, яка описує з'єднання з каналом зв'язку TCP/IP.

Клас System.Net.Sockets.Socket. Клас Socket виконує важливу роль в мережевому програмуванні, забезпечуючи функціонування як клієнту, так і серверу. Головним чином методи цього класу виконують необхідні перевірки, пов'язані з безпекою, зокрема перевіряють дозволи системи безпеки.

Перш ніж використовувати клас Socket, потрібно розглянути деякі важливі складові класу System.Net.Sockets.Socket.

*AddressFamily* – дає сімейство адрес сокету.

*Available* – повертає об'єм доступних для читання даних.

*Blocking* – дає або встановлює значення, що показує, чи знаходиться сокет у блокованому режимі.

*Connected* – повертає значення, що інформує, чи з’єднаний сокет з віддаленим хостом.

*LocalEndPoint* – визначає локальну кінцеву крапку.

*ProtocolType* – визначає тип протоколу сокета.

*RemoteEndPoint* - повертає віддалену кінцеву точку сокета.

*SocketType* – визначає тип сокета.

Розглянемо важливі методи класу System.Net.Sockets.Socket.

*Accept()* – створює новий сокет для обробки вхідного запиту на з'єднання

*Bind()* – пов'язує сокет з локальною кінцевою крапкою для очікування вхідних запитів на з'єднання.

*Close()* – примушує сокет закритися.

*Connect()* – встановлює з'єднання з віддаленим хостом.

*GetSocketOption()* – повертає значення SocketOption.

*Listen()* – поміщає сокет в режим прослуховування, цей метод призначений тільки для серверних додатків.

*Receive()* – одержує дані від'єднаного сокета. Перевіряє статус одного або декількох сокетів.

*Send()* – відправляє дані під'єднаному сокету.

*Shutdown()* – забороняє операції відправки і приїому даних за допомогою сокета.

### 10.3. Задача 1. Отримання адреси машини з використанням DNS

Для цього використовується класс System.Net.Dns який надає методи, що повертають інформацію про мережеві адреси, підтримувані пристроєм в локальній мережі. Якщо у пристрою локальної мережі є більше однієї мережевої адреси, клас Dns повертає інформацію про всі мережеві адреси.

```
static void Main(string[] args)
{
        string name = (args.Length < 1) ? Dns.GetHostName() : args[0];
        try
        {
           IPAddress[] addrs = Dns.Resolve(name).AddressList;
           foreach (IPAddress addr in addrs)
              Console.WriteLine("{0} | {1}", name, addr);
        }
```

```
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
```

Якщо запустити програму зі значенням параметра «www.rambler.ru» отримуємо результат, показаний на рис.70.

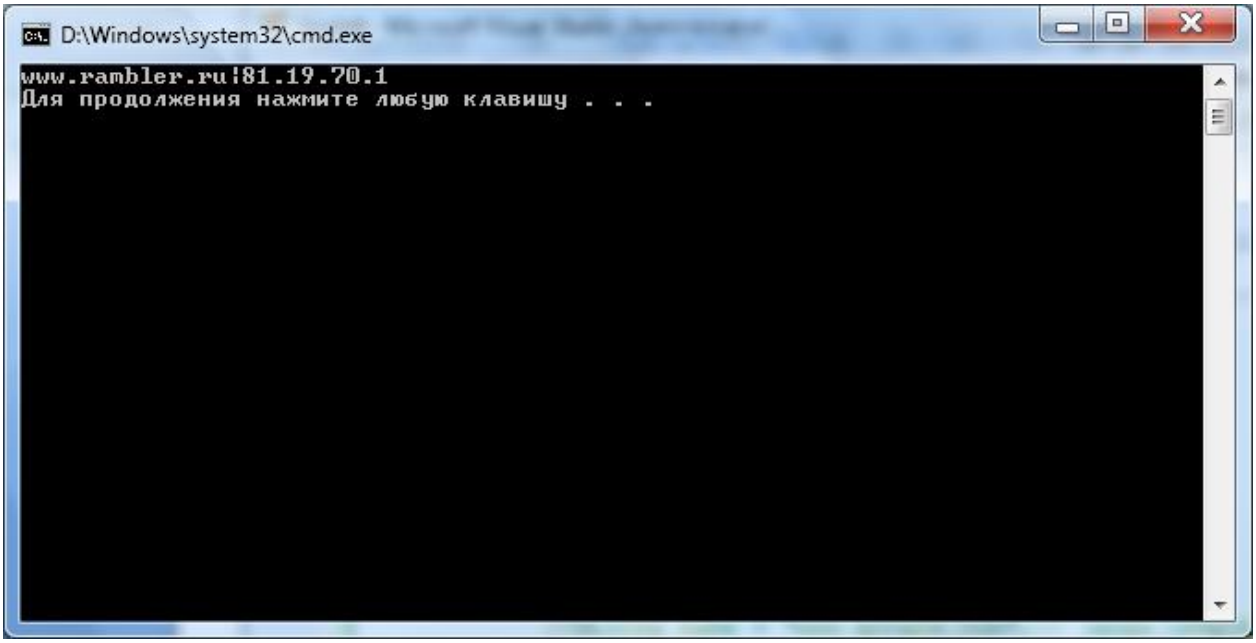



Рис.70. Результат виконання програми

### 10.4.Задача 2. Створення додатку з потоковим типом взаємодії

Класичним прикладом використання механізму сокетів є побудова системи клієнт-серверної архітектури, за якою один додаток грає роль серверу, що виконує запроси другого додатку.

Існує два підходи використання сокетів в додатках: синхронний та асинхронний. З початку розглянемо синхронний варіант організації взаємодії.

#### 10.4.1.Синхронна взаємодія

Цей варіант означає, що виконання потоку, який відповідає за взаємодію блокується, поки сервер не дасть згоди на з'єднання з клієнтом.

Наступний додаток демонструє тип простого серверу, що відповідає клієнту. Клієнт завершує з'єднання, відправляючи серверу повідомлення <TheEnd>.

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
namespace SocketTest
{
   public class Server
   {
     public void Run()
      {
        //Встановлюємо для сокету локальну точку
        string hostName = Dns.GetHostName();
        IPAddress[] ipHostAddressList = Dns.Resolve(hostName).AddressList;
        IPAddress ipAddr = ipHostAddressList[0];
        IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 11000);
        // створюємо сокет Tcp/IP
        Socket serverSocket = new Socket(AddressFamily.InterNetwork,
                                         SocketType.Stream,
                                         ProtocolType.Tcp);
        //Назначаємо сокет локальній кінцевій точці
        //і слухаємо вхідні сокети
        try
        {
           serverSocket.Bind(ipEndPoint);
           string header = "Server [" + serverSocket.LocalEndPoint.ToString() + "]: ";
           serverSocket.Listen(10);
           Console.WriteLine("Server has started at {0}.", ipEndPoint);
           // починаємо зєднання
           while (true)
           {
             // програма призупиняється, очікуючи вхідне повідомлення.
```

```
                Socket clientSocket = serverSocket.Accept();
                string data = null;
               //Дочекались клієнта, який намагається звязатись з сервером
                while (true)
                {
                   byte[] bytes = new byte[1024];
                   int bytesRec = clientSocket.Receive(bytes);
                   data += Encoding.ASCII.GetString(bytes, 0, bytesRec);
                   if (data.IndexOf("<TheEnd>") > -1)
                      break;
                }
                //відображаєсо прийняті дані
                Console.WriteLine(header + " Received from [{0}]: " +
                                  data, clientSocket.RemoteEndPoint);
                string theReply = "Server's answer";
                byte[] msg = Encoding.ASCII.GetBytes(theReply);
                clientSocket.Send(msg);
                clientSocket.Shutdown(SocketShutdown.Both);
                clientSocket.Close();
             }
          }
          catch (Exception e)
          {
             Console.WriteLine(e.ToString());
          }
       }
    }
}
```

Перший крок полягає у встановленні для сокета локальної точки (адреси процесу).

Далі створєтся потоковий сокет.

```
Socket slistener = new Socket(AddressFamily.InterNetwork,
        SocketType.Stream, ProtocolType.Tcp);
```

Наступним кроком є зв'язок сокета з адресою. Для цього використовується метод Bind(), який пов'язує сокет з локальною кінцевою крапкою. Викликати метод Bind( ) треба до будь-яких спроб звернення до методів Listen( ) і Accept( ).

Після цього, скориставшися методом Listen (), переводимо сокет в стан прослуховування. В стані прослуховування сокет чекатиме вхідні спроби з'єднання.

```
sListener.Listen(10);
```

У параметрі визначається розділ (backlog), який вказує максимальне число з'єднань, що чекають обробки в черзі. У приведеному коді значення параметра допускає накопичення в черзі до десяти з'єднань.

В стані прослуховування треба бути готовим дати згоду на з'єднання з клієнтом, для чого використовується метод Accept(). За допомогою цього методу здійснюється з'єднання клієнту і завершується встановлення зв'язку імен клієнту і серверу. Метод Accept() блокує потік визиваючої програми до здійснення з'єднання.

Метод Accept() витягує з черги чекаючих запитів перший запит на з'єднання і створює для його обробки новий сокет. Хоча новий сокет і створений, первинний сокет продовжує слухати і може використовуватися при багатопотоковій обробці для прийому декількох запитів на з'єднання від клієнтів. Ніякий серверний додаток не повинен закривати слухаючий сокет. Він повинен продовжувати працювати разом з сокетами, створеними методом Accept для обробки вхідних запитів клієнтів.

Після встановлення з'єднання, можна відправляти і одержувати повідомлення, використовуючи методи Send() і Receive( ) класу Socket.

Метод Send() записує витікаючі дані сокету, з яким встановлено з'єднання. Метод Receive() прочитує вхідні дані в потоковий сокет. При використовуванні системи, основаної на TCP, перед виконанням методів Send() і Receive() між сокетами повинно бути встановлено з'єднання. Точний протокол між двома взаємодіючими сокетами повинен бути визначений завчасно,

щоб клієнтський і серверний додатки не блокували один одного, не знаючи, хто повинен відправити свої дані першим.

Метод Receive() зчитує данні із сокету і записує їх до масиву байтів, переданого в якості аргументу. Метод повертає число, яке дорівнює кількості отриманих байтів.

При виході з циклу відправляємо масив байтів клієнту :

```
string theReply = "Server's answer";
byte[] msg = Encoding.ASCII.GetBytes(theReply);
handler.Send(msg);
```

Коли обмін між сервером і клієнтом закінчується, необхідно закрити зєднання методом Close().

Функції, які використовуються для створення додатку-клієнта, в деякій мірі нагадують серверний додаток. Як і для серверу, використовуються ті ж методи для визначення кінцевої крапки, створення екземпляра сокету, відправки і отримання даних і закриття сокету.

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
namespace SocketTest
{
   public class Client
   {
      public void Run()
      {
         byte[] bytes = new byte[1024];
         //зєднання з віддаленим пристроєм.
         try
         {
            string header;
            //встановлюємо віддалену кінцеву крапку для сокета.
            string name = Dns.GetHostName();
            IPAddress[] ipHostAddressList = Dns.Resolve(name).AddressList;
```

```
                IPAddress ipAddr = ipHostAddressList[0];
                IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 11000);
                Socket sender = new Socket(AddressFamily.InterNetwork,
                SocketType.Stream, ProtocolType.Tcp);
                IPEndPoint localIP = new IPEndPoint(ipAddr, 0);
                sender.Bind(localIP);
                header = "Client ["+ sender.LocalEndPoint.ToString()+"]: ";
                // зєднуємо сокет з віддаленою кінцевою точкою
                sender.Connect(ipEndPoint);
                string theMessage = "This is a test";
                byte[] msg = Encoding.ASCII.GetBytes(theMessage + " <TheEnd>");
                Console.WriteLine(header + "Send: " + theMessage + " <TheEnd>");
                //відправляємо дані через сокет
                sender.Send(msg);
                //отримуємо відповідь від віддаленого пристрою
                sender.Receive(bytes);
                Console.WriteLine(header + "Recieved from server: {0}",
Encoding.ASCII.GetString(bytes));
                //звільняємо сокет.
                sender.Shutdown(SocketShutdown.Both);
                sender.Close();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
        }
    }
}
```

В даному коді присутній  новий метод Connect (), що використовується для з'єднання з віддаленим сервером.

Розглянемо механізм тестування даних модулів. Для цього створюємо додаткову програму:

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
namespace SocketTest
{
   class Program
   {
      private static Server server = new Server();
      private static Client client = new Client();
      static void Main(string[] args)
      {
         Thread serverThread = new Thread(startServer);
         serverThread.IsBackground = true;
         serverThread.Start();
         Console.WriteLine("Press ENTER to test interaction...");
         if (Console.ReadKey().Key == ConsoleKey.Enter) ;
         {
            Console.WriteLine("Press ESC to exit");
            while (true)
            {
               if (Console.ReadKey().Key == ConsoleKey.Escape)
               {
                  break;
               }
               else
               {
                  client.Run();
               }
            }
         }
      }
      private static void  startServer()
      {
```

```
            server.Run();
        }
    }
}
```

Результати дії програми показані на рис.71.

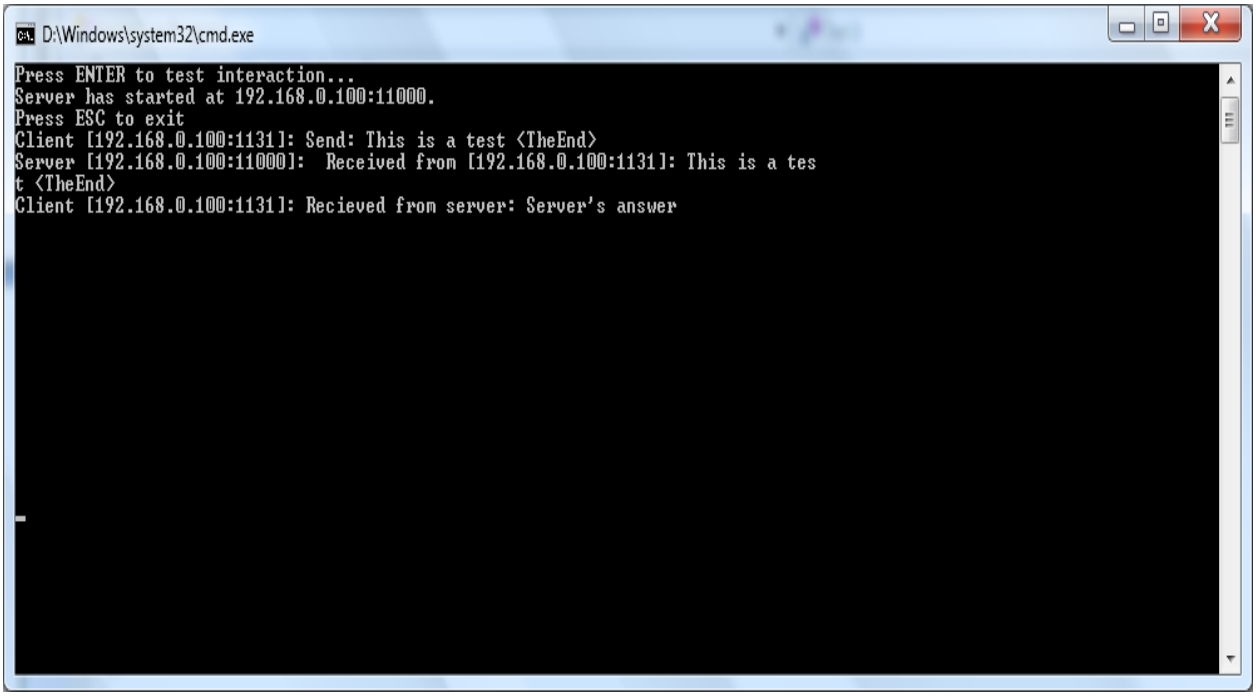



Рис.71. Результати виконання програми

### 10.4.2. Асинхронний варіант взаємодії (.NET 2.0)

Існує дві проблеми з синхронною взаємодією.

1. Виклик функції Receive блокує виконання програми до того моменту доки не будуть отримані дані.

2. У разі отримання даних не відомо, коли здійснювати виклик наступного разу. Тому використовується механізм постійного запиту (polling).

Хоча подолання цих недоліків можливо шляхом використання багатопотокового програмування (головна програма відкриває новий потік для кожної взаємодії), такий підхід також має певні недоліки.

Операційна система Windows (Windows NT /2000 /XP) пропонує механізм Completion Port IO model для здійснення асинхронного вводу-виводу

(overlapped ( asynchronous) IO). Деталі доступні на сайтах http://msdn.microsoft.com/en-us/library/aa365198(VS.85).aspx та http://technet. microsoft.com/en-us/sysinternals/bb963891.aspx).

Механізм Completion Port model зв'язаний з усіма варіантами IO, включаючи читання/ запис даних до файлу, операціями послідовної передачі даних, socket-програмуванням.

В .NET framework асинхронний варіант взаємодії здійснюється на базі класу Socket. Клас Socket містить методи, що використовують клас AsyncCallback для виклику процедур завершення (completion methods), коли мережеві функції (встановлення зв'язку, прийом, відправлення) завершили свою роботу. Клас AsyncCallback дозволяє здійснювати виклик асинхронної мережевої функції та призначати обробник, який буде викликатися після завершення асинхронної функції [Krowczyk, 2005].

Такий спосіб дозволяє додатку виконувати операції одночасно з виконанням або очікуванням виконання мережевих операцій (overlapping).

Асинхронні методи класу Socket розділяються на два типи: Begin-методи, що починають виконання мережевої функції та визначають метод-обробник, що є аргументом методу AsyncCallback; End-методи, що завершують виконання мережевої функції після виклику обробника AsyncCallback.

Розглянемо основні методи та типову програму-сервер.

BeginAccept - асинхронний метод (non−blocking method), який викликає обробник, визначений делегатом та повертає управління основному потоку. EndAccept повертає об'єкт сокету для здійснення взаємодії. Наприклад , процедура serverSocket.BeginAccept(new AsyncCallback (OnClientConnect), null);

викликає метод-обробник OnClientConnect при запрошенні взаємодії від клієнтського процесу. Другим параметром може бути любий об'єкт, що передається в полі AsyncState параметра типу інтерфейсу IAsyncResult.

```
public void OnClientConnect(IAsyncResult asyn)
{
```

```
            try
            {
                clientSocket = serverSocket.EndAccept(asyn);
                ProcessClient(clientSocket); //processing accepted socket
                serverSocket.BeginAccept(new AsyncCallback(OnClientConnect),
                 null);
                      }
            catch (ObjectDisposedException)
            {
                System.Diagnostics.Debugger.Log(0, "1", "\n OnClientConnection: Socket has
been closed\n");
            }
            catch (SocketException se)
            {
                Console.WriteLine(se.Message);
            }
        }
```

Метод BeginReceive забезпечує асинхронний прийом даних

```
public IAsyncResult BeginReceive( byte[] buffer,
int offset,
int size,
SocketFlags socketFlags,
AsyncCallback callback,
object state );
```

Тут buffer – буфер даних ,

callback – callback-фукція (delegate), що завжди викликається в разі надходження та прийому даних,

object – об’єкт классу ( може дорівнювати null ) .

Callback-функція має бути визначена як void та мати вхідним параметром IAsyncResult ar.

Параметр IAsyncResult характеризує статус операції прийому даних і містить:

AsyncState – об’єкт, що передається параметром object,

IsComplete – процедура, що вказує на завершення операції.

Розглянемо приклад організації типової взаємодії з використанням описаних методів.

Вводимо тип, що відповідає за репрезентацію пакету сокета, який має два поля: сокет через який здійснюється взаємодія, буфер в який приймаються дані.

```
public class SocketPacket
{
   public System.Net.Sockets.Socket CurrentSocket;
   public byte[] DataBuffer = new byte[100];
}
```

Вводимо поля

```
private string sData = "";
//Data
public AsyncCallback receiverCallBack; //Callback function
```

Здійснюємо рослуховування сокету

```
public void OnClientConnect(IAsyncResult asyn)
{
   try
   {
      clientSocket = serverSocket.EndAccept(asyn);
      // Let the worker Socket do the further processing for the
      // just connected client
      ProcessClient(clientSocket);
      // Now increment the client count
      serverSocket.BeginAccept(
       new AsyncCallback(OnClientConnect), null);
   }
   catch (ObjectDisposedException)
   {
      System.Diagnostics.Debugger.Log(0, "1", "\n
```

```
            OnClientConnection: Socket has been closed\n");
        }
        catch (SocketException se)
        {
            Console.WriteLine(se.Message);
        }
    }
```

Обробка з’єднання з клієнтом

```
    public void ProcessClient(System.Net.Sockets.Socket socket)
    {
        try
        {
            if (receiverCallBack == null)
            {
                receiverCallBack = new AsyncCallback(OnDataReceived);
            }
            SocketPacket socketPacket;
            socketPacket = new SocketPacket();
            socketPacket.CurrentSocket = socket;
            socket.BeginReceive(socketPacket.DataBuffer, 0,
                                socketPacket.DataBuffer.Length,
                                SocketFlags.None,
                                receiverCallBack,
                                socketPacket);
        }
        catch (SocketException se)
        {
            Console.WriteLine(se.Message);
        }
    }
```

Обробник перевіряє отримані дані на присутність маркеру кінця пакету (<<EOP>>) і у разі його відсутності продовжує прийом даних. У разі успішного прийому – надсилає повідомлення «АСК».

```
    public void OnDataReceived(IAsyncResult asyn)
```

```
{
   try
   {
      SocketPacket socketPacket = (SocketPacket)asyn.AsyncState;
      int bytesRead = socketPacket.CurrentSocket.EndReceive(asyn);
      if (bytesRead > 0)
      sData += Encoding.ASCII.GetString(socketPacket.DataBuffer,
                                                0, bytesRead);
      if (sData.IndexOf("<<EOP>>") == -1)
      {
         ProcessClient(socketPacket.CurrentSocket);
      }
      else
      {
          Console.WriteLine(sData);
          sData = "";
          byte[] serverAnswer =
          System.Text.Encoding.ASCII.GetBytes("ACK");
          socketPacket.CurrentSocket.Send(serverAnswer);
          socketPacket.CurrentSocket.Close();
          socketPacket.CurrentSocket = null;
      }
   }
   catch (ObjectDisposedException)
   {
      System.Diagnostics.Debugger.Log(0,
              "1",
              "\nOnDataReceived: Socket has been closed\n");
   }
   catch (SocketException se)
   {
      Console.WriteLine(se.Message);
   }
}
```

Розглянемо задачу часового лімітування операції прийому даних. Слід відзначити, що властивість ReceiveTimeout в даному випадку не використовується. Для виконання даної задачі використовується властивість AsyncWaitHandle - об'єкту результата асинхронної операції.

Метод WaitOne класу AsyncWaitHandle блокує виконання функції (потоку) до виконання визначених умов (доки WaitHandle не отримує сигнал, що визначений інтервал пройшов). Другий параметр вказує на можливість виходу з домену синхронізації перед очикуванням.Так, наприклад, код

```
IAsyncResult result =
socket.BeginReceive(socketPacket.DataBuffer, 0,
 socketPacket.DataBuffer.Length,
 SocketFlags.None,
 pfnWorkerCallBack,
 socketPacket);
//here we can wait for 3 seconds for completion of receive operation
if (result.AsyncWaitHandle.WaitOne(3000, false) == false)
{
   throw new Exception("Timeout reading from socket!");
}
```

блокує виконання функції на 3 секунди.

### 10.4.3. Приклад тестування паралельної обробки «асинхронного» серверу

В наведеному прикладі виконується запуск трьох потоків: один сервер, два клієнта. Після цього додаток очікує натискання клавіші Enter. Якщо клавіша натиснута – виконується запуск двох клієнтів. Далі очікується натиск клавіші і при натисненій клавіші Escape здійснюється вихід.

Слід відзначити, що серверний додаток очікує 3 секунди на виконання операції. Таким чином, якщо обробка одночасно запущених сокетів буде проходити з інтервалом в 3 секунди – це послідовна обробка, якщо ні – паралельна.

```
class Program
 {
   private static AsyncTcpServer asynchTcpServer = new AsyncTcpServer();
   private static Client client = new Client();
   static void Main(string[] args)
    {
       RunTcpAsynchServer();
    }
  private static void RunTcpAsynchServer()
    {
       Thread serverThread = new Thread(startAsynchServer);
       serverThread.IsBackground = true;
       Thread clientThread = new Thread(startClient);
       clientThread.IsBackground = true;
       Thread clientThread2 = new Thread(startClient);
       clientThread2.IsBackground = true;
       serverThread.Start();
       Console.WriteLine("Press ENTER to test interaction...");
       if (Console.ReadKey().Key == ConsoleKey.Enter) ;
       {
          clientThread.Start();
          clientThread2.Start();
          Console.WriteLine("Press ESC to exit");
          while (true)
          {
             if (Console.ReadKey().Key == ConsoleKey.Escape)
             {
                break;
             }
             else
             {
                client.Run();
             }
          }
       }
```

```
        }
        private static void startAsynchServer()
        {
            asynchTcpServer.Run();
        }
        private static void startClient()
        {
                client.Run();
        }
    }
```

### 10.4.4. Асинхронний варіант взаємодії (.NET 3.5)

Як вже було вказано, особливістю класу Socket версії 2.0 є виключення використання процесору з процесу обробки операції вводу/виводу (single socket I/O operation) та виділення обєктів, щоб надати можливість одночасної обробки значного числа сокетів.

В середовищі .NET Framework 3.5 надана можливість більш ефективного керування великою кількістю об'єктів, що одночасно обробляються (overlapped objects) на рівні CLR. Такі об'єкти (overlapped objects) ефективно використовують структуру Windows OVERLAPPED, яка застосується для керування асинхронними операціями вводу/виводу (asynchronous I/O opera?ions). Таким чином існує тільки один об'єкт-екземпляр (Overlapped object instance) для кожної операції асинхронного вводу/виводу (Socket async?ronous I/O operation in progress). Така організація дозволяє мати більше 60000 сокетів у стані з'єднання (connected sockets) , очкування асинхронного прийому даних (asynchronous receive I/O operation).

Класи Socket в версії 2.0 використовують Windows I/O Completion Ports для виконання асинхронної операції вводу/виводу. Це дозволяє додатку використовувати велике число відкритих сокетів, що дуже актуально для масштабування систем. В .NET Framework реалізований клас System.Threading.ThreadPool, який пропонує потокі (completion threads), які

читають порти (Completion Ports) та забезпечують виконання асінхроних операцій вводу/виводу.

В версії 3.5 .NET Framework особлива увага сфокусована на виключенні накладних витрат на передачах коду (code paths) між процедурами читання порту (Completion Port) та викликом делегату завершення операції або генерації події завершення операції з повертанням об'єкту IAsyncResult.

Важливо зазначити, що для кожної операції створюється об'єкт IAsyncResult, який не може повторно використовуватися, що викликає проблеми виділення та відповідно видалення ресурсів для визначеного об'єкту. Для вирішення проблеми нова реалізація пропонує інший шаблон методів виконання асинхронного вводу/виводу з застосуванням сокетів. Такий шаблон не вимагає застосування об'єкту контексту операції для кожної операції сокету [http://msdn.microsoft.com/en-us/magazine/ cc163356.aspx].

Таким чином до класу Socket були введени нові методи та надано механизм асинхронної обробки, оснований на подіях. В версії 2.0 використовується наступний код для асинхронної відсилки даних з застосуванням сокету:

```
void OnSendCompletion(IAsyncResult ar) { }
IAsyncResult ar = socket.BeginSend(buffer, 0, buffer.Length,
SocketFlags.None, OnSendCompletion, state);
```

В новій версії використовується інша модель:

```
void OnSendCompletion(object src, SocketAsyncEventArgs sae) { };
SocketAsyncEventArgs sae = new SocketAsyncEventArgs();
sae.Completed += OnSendCompletion;
sae.SetBuffer(buffer, 0, buffer.Length);
socket.SendAsync(sae);
```

Маємо наступні відмінності:

1. Застосування об'єкту типу SocketAsyncEventArgs, що формує контекст операції, замість об'єкту IAsyncResult.

2. Всі параметри операції з сокетом визначаються властивостями та методами об’єкту SocketAsyncEventArg. Статус завершення операції також визначається через властивості об’єкту SocketAsyncEventArgs.
3. Замість AsyncCallback функції використовується підпис на подію.

Нижче приведена програма серверу, що базується на застосуванні асинхронних сокетів. Особливу увагу слід звернути на блоки перевірки умови типу в обробниках події Completed :

```
if (e.LastOperation == SocketAsyncOperation.Accept)
{ … }
```

Як ми бачимо з наведеного коду, на подію підписано два обробника:

```
socketAsyncEventArgs.Completed +=
        new EventHandler<SocketAsyncEventArgs>(OnClientConnect);
```

та

```
socketAsyncEventArgs.Completed +=
        new EventHandler<SocketAsyncEventArgs>(Receive_Completed）.
```

При цьому в блоці коду обробника OnClientConnect є визов методу ProcessClient, в тілі якого здійснюється друга підписка та виклик методу прийому. Така ситуація без аналізу того, яка операція викликала генерацію події (прийома або підключення сокету), приводить до потері даних (відмінить цю перевірку та проаналізуйте результат), наприклад :

```
namespace SocketTest
{
  public class AsyncEventTcpServer
  {
    readonly Socket serverSocket = new Socket(AddressFamily.InterNetwork,
    SocketType.Stream,ProtocolType.Tcp);
    private Socket[] _workerSocketArray = new Socket[10];
    private int _clientCount = 0;
    public void Run()
    {
      string hostName = Dns.GetHostName();
      IPAddress[] ipHostAddressList = Dns.Resolve(hostName).AddressList;
```

```
    IPAddress ipAddr = ipHostAddressList[0];
    IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 11000);
    try
    {
       serverSocket.Bind(ipEndPoint);
       string header = "Server [" +
             serverSocket.LocalEndPoint.ToString() + "]: ";
       serverSocket.Listen(10);
       Console.WriteLine("Server has started at {0}.", ipEndPoint);
       StartAccept(this, null);
    }
    catch (Exception e)
    {
       Console.WriteLine(e.ToString());
    }
}
private void StartAccept(object sender,
                    SocketAsyncEventArgs  socketAsyncEventArgs)
{
    if (socketAsyncEventArgs == null)
    {
       socketAsyncEventArgs = new SocketAsyncEventArgs();
       socketAsyncEventArgs.Completed +=
         new EventHandler<SocketAsyncEventArgs>(OnClientConnect);
    }
    socketAsyncEventArgs.AcceptSocket = null;
    try
    {
       serverSocket.AcceptAsync(socketAsyncEventArgs);
    }
    catch (Exception ex)
    {
       Debug.WriteLine(string.Format("Exception SocketListener:{0}",
     ex.GetType().Name));
    }
```

```
    finally
    {
        Debug.WriteLine("Exit  SocketAsyncListener");
    }
}
public void OnClientConnect(object sender, SocketAsyncEventArgs e)
{
    try
    {
        if (e.LastOperation == SocketAsyncOperation.Accept)
        {
            if (e.SocketError == SocketError.OperationAborted)
            {
              Debug.WriteLine(string.Format("Info
               SocketAsyncListener::EndAccept listener#{0} is closed",
                                        sender.GetHashCode()));
               e.Dispose();
               Debug.WriteLine(
                  string.Format("Exit
 SocketAsyncListener::EndAccept SocketAsyncEventArgs#{0} is disposed",
                              e.GetHashCode()));
               return;
            }
            Socket listener = (Socket) sender;
            _workerSocketArray[_clientCount] = e.AcceptSocket;
            String str = String.Format("Client # {0} connected", _clientCount + 1);
            Console.WriteLine("{0} + {1}", str, e.AcceptSocket.RemoteEndPoint);
            ++_clientCount;
            ProcessClient(e);
            StartAccept(listener, null);
        }
    }
    catch (ObjectDisposedException)
    {
```

```
                System.Diagnostics.Debugger.Log(0, "1", "\n OnClientConnection: Socket has
been closed\n");
            }
            catch (SocketException se)
            {
                Console.WriteLine(se.Message);
            }
        }
        public class SocketPacket
        {
            public System.Net.Sockets.Socket CurrentSocket;
            public byte[] DataBuffer = new byte[100];
        }
        public void ProcessClient(SocketAsyncEventArgs socketAsyncEventArgs)
        {
            try
            {
                SocketPacket socketPacket;
                socketPacket = new SocketPacket();
                socketPacket.CurrentSocket =
                            socketAsyncEventArgs.AcceptSocket;
                try
                {
                    socketAsyncEventArgs.UserToken = socketPacket;
                    socketAsyncEventArgs.SetBuffer(new byte[100], 0, 100);
                    socketAsyncEventArgs.Completed +=
                    new EventHandler<SocketAsyncEventArgs>(Receive_Completed);
                    socketPacket.CurrentSocket.ReceiveAsync(socketAsyncEventArgs);
                }
                catch (Exception ex)
                {
                    string exc = ex.Message;
                    byte[] serverAnswer =
                        System.Text.Encoding.ASCII.GetBytes("timeout");
                    socketPacket.CurrentSocket.Send(serverAnswer);
```

```
        }
    }
    catch (SocketException se)
    {
        Console.WriteLine(se.Message);
    }
}
private void Receive_Completed(object sender, SocketAsyncEventArgs e)
 {
    string sData = "";
    SocketPacket socketPacket = e.UserToken as SocketPacket;
    if (e.LastOperation == SocketAsyncOperation.Receive)
    {
        if (e.BytesTransferred == 0 ||
            e.SocketError != SocketError.Success ||
            socketPacket == null)
        {
            socketPacket.CurrentSocket.Close();
            return;
        }
        socketPacket.DataBuffer = e.Buffer;
        sData += Encoding.ASCII.GetString(e.Buffer,
                        e.Offset, e.BytesTransferred);
        if (sData.IndexOf("<TheEnd>") == -1)
        {
            ProcessClient(e);
        }
        else
        {
            Console.WriteLine("Server: " + sData);
            sData = "";
            byte[] serverAnswer =
                System.Text.Encoding.ASCII.GetBytes("answer");
            socketPacket.CurrentSocket.Send(serverAnswer);
            socketPacket.CurrentSocket.Close();
```

```
                socketPacket.CurrentSocket = null;
            }
        }
    }
    void CloseSockets()
    {
        if (serverSocket != null)
        {
            serverSocket.Close();
        }
        for (int i = 0; i < _clientCount; i++)
        {
            if (_workerSocketArray[i] != null)
            {
                _workerSocketArray[i].Close();
                _workerSocketArray[i] = null;
            }
        }
    }
}
```

### 10.5. Задача 3. Організація взаємодії з використанням протоколу UDP

Як вже було сказано, протокол UDP не вимагає зєднання – дані можуть бути надіслані не одному, а множині отримувачів використовуючи при цьому один сокет. Для организації взаємодії використовується класс System.Net.Sockets.UdpClient, в конструктор якого передається адреса віддаленого хоста та номер порту у разі клієнтського додатку, або номер порту в разі серверного.

```
UdpClient udpc = new UdpClient(2055);
```

та

```
string hostName = Dns.GetHostName();
    IPAddress[] ipHostAddressList = Dns.Resolve(hostName).AddressList;
    IPAddress ipServerAddress = ipHostAddressList[0];
```

```
UdpClient udpc = new UdpClient(ipServerAddress.ToString(), 2055).
```

Отримання даних здійснюється з використанням методу Receive:

```
IPEndPoint ep = null;
byte[] data = udpc.Receive(ref ep);
```

де data – байтовий масив, в якому зберігаються отримані дані,

ep – адреса відправника.

Відправка даних здійснюється з використанням методу Send. Якщо адреса віддаленого хосту була задана в конструкторі, використовується виклик методу

```
udpc.Send(sdata, sdata.Length);
```

де data – байтовий масив.

Інакше, в метод передається адреса процесу отримувача

```
udpc.Send(sdata, sdata.Length, ep);
```

де IPEndPoint ep – адреса отримувача.

Розглянемо задачу віддаленої ідентифікації користувача.

Серверна програма має вигляд:

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
namespace SocketTest
{
  public class UDPServer
  {
    public void Run()
    {
      UdpClient udpc = new UdpClient(2055);
      Console.WriteLine("Server started on port 2055");
      IPEndPoint ep = null;
      string PID;
      string status = "not identified";
      while (true)
      {
```

```
            byte[] rdata = udpc.Receive(ref ep);
            PID = Encoding.ASCII.GetString(rdata);
            //Some logic to identify
            if (PID == "12345")
               status = "identified";
            byte[] sdata = Encoding.ASCII.GetBytes(status);
            udpc.Send(sdata, sdata.Length, ep);
         }
      }
   }
}
```

Клієнтська програма :

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
namespace SocketTest
{
   public class UDPClient
   {
      public void Run()
      {
         string hostName = Dns.GetHostName();
         IPAddress[] ipHostAddressList = Dns.Resolve(hostName).AddressList;
         IPAddress ipServerAddress = ipHostAddressList[0];
         UdpClient udpc = new UdpClient(ipServerAddress.ToString(), 2055);
         IPEndPoint ep = null;
         while (true)
         {
            Console.Write("Enter your personal identifier: ");
            string PID = Console.ReadLine();
            if (PID == "" )
            {
               //Console.Write("Press ESC to quit. ");
               break;
```

```
            }
            //
            byte[] sdata = Encoding.ASCII.GetBytes(PID);
            udpc.Send(sdata, sdata.Length);
            byte[] rdata = udpc.Receive(ref ep);
            string status = Encoding.ASCII.GetString(rdata);
            Console.WriteLine(status);
            if (status == "identified")
            {
               break;
            }
         }
      }
   }
}
```

Тестова програма має вигляд:

```
    private static UDPServer udpserver = new UDPServer();
     private static UDPClient udpclient = new UDPClient();
     static void Main(string[] args)
     {
        Thread serverThread = new Thread(startUDPServer);
        serverThread.IsBackground = true;
        serverThread.Start();
        Console.WriteLine("Press ENTER to test interaction...");
        if (Console.ReadKey().Key == ConsoleKey.Enter) ;
        {
           udpclient.Run();
        }
     }
     private static void startUDPServer()
     {
        udpserver.Run();
     }
```

Результати тесту показан на рис.72.

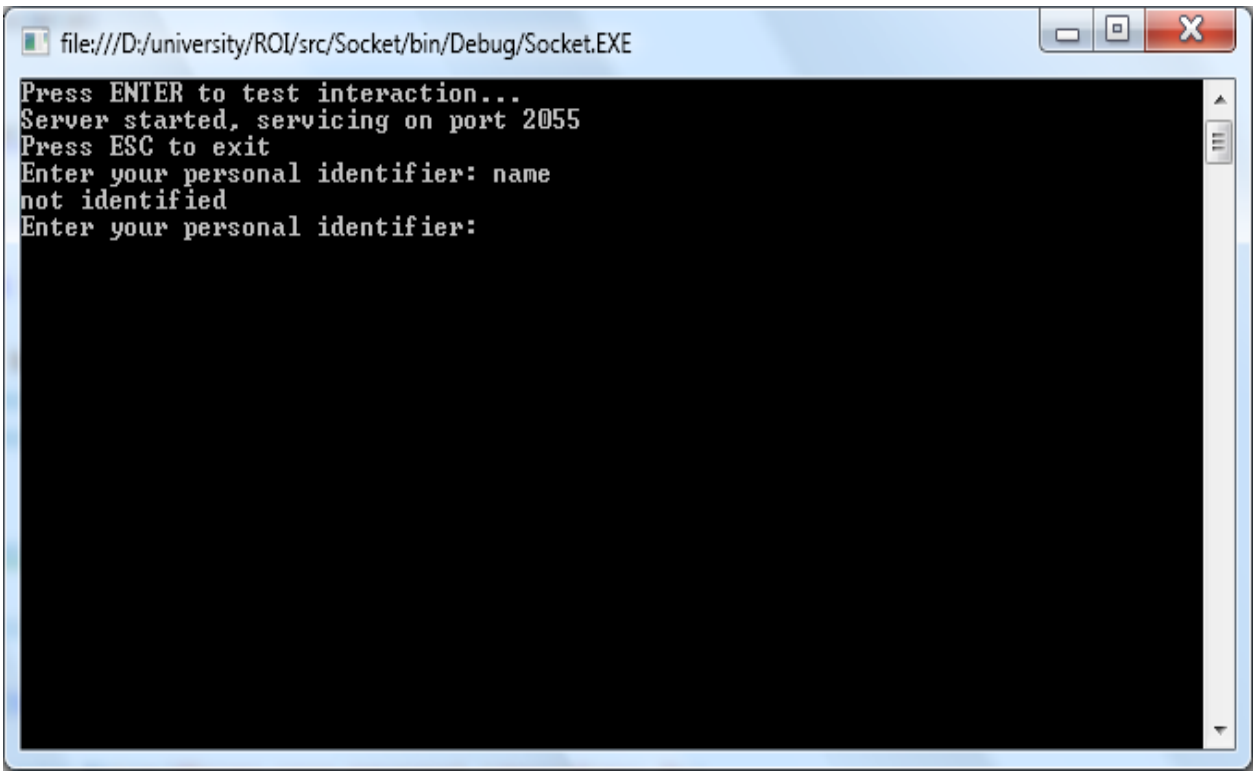



Рис.72. Результати виконання тестової програми

Розглянемо приклад посилки однієї датаграми декількам отримувачам. Для цього отправувач відправляє пакети на адресу в діапазоні 224.0.0.1 - 239.255.255.255(Class D address group). Отримувачі можуть увійти в цю групу (join the group) та почати отримувати пакети. Вход в групу здійснюється методом JoinMulticastGroup(addr).

Розглянемо приклад публікації цін на товар «product» сервером, що здійснюється кожні 5 секунд. Для цього на адресу 230.0.0.1 відправляється датаграма, яку отримують клієнти, що входять у групу.

Код серверу :

```
public class UDPPublisher
{
   private string product = "Product";
   public void Run()
   {
      UdpClient publisher = new UdpClient("230.0.0.1",8899);
       Console.WriteLine("Publishing stock prices to 230.0.0.1:8899");
      Random gen = new Random();
```

```
        while(true)
        {
            double price = 400*gen.NextDouble()+100;
            string msg = String.Format("{0} {1:#.00}",product,price);
            byte[] sdata = Encoding.ASCII.GetBytes(msg);
            publisher.Send(sdata,sdata.Length);
            System.Threading.Thread.Sleep(5000);
        }
    }
}
```

Код клієнту :

```
class UDPSubscriber
{
    public void Run()
    {
    UdpClient subscriber = new UdpClient(8899);
    IPAddress addr = IPAddress.Parse("230.0.0.1");
    subscriber.JoinMulticastGroup(addr);
    IPEndPoint ep = null;
        for(int i=0; i<10;i++){
        byte[] pdata = subscriber.Receive(ref ep);
        string price = Encoding.ASCII.GetString(pdata);
        Console.WriteLine(price);
        System.Threading.Thread.Sleep(3000);
        }
       subscriber.DropMulticastGroup(addr);
    }
}
```

Код тестеру :

```
    private static UDPPublisher udppublisher = new UDPPublisher();
    private static UDPSubscriber udpsubscriber = new UDPSubscriber();
    static void Main(string[] args)
    {
       Thread serverThread = new Thread(startUDPPublisher);
```

```
            serverThread.IsBackground = true;
            serverThread.Start();
            Console.WriteLine("Press ENTER to test interaction...");
            if (Console.ReadKey().Key == ConsoleKey.Enter) ;
            {
                udpsubscriber.Run();
                Console.WriteLine("Press ESC to exit");
                while (true)
                {
                    if (Console.ReadKey().Key == ConsoleKey.Escape)
                    {
                        break;
                    }
                }
            }
        }
        private static void startUDPPublisher()
        {
            udppublisher.Run();
        }
```

Результат цих дій має наступний вигляд (рис.73) :

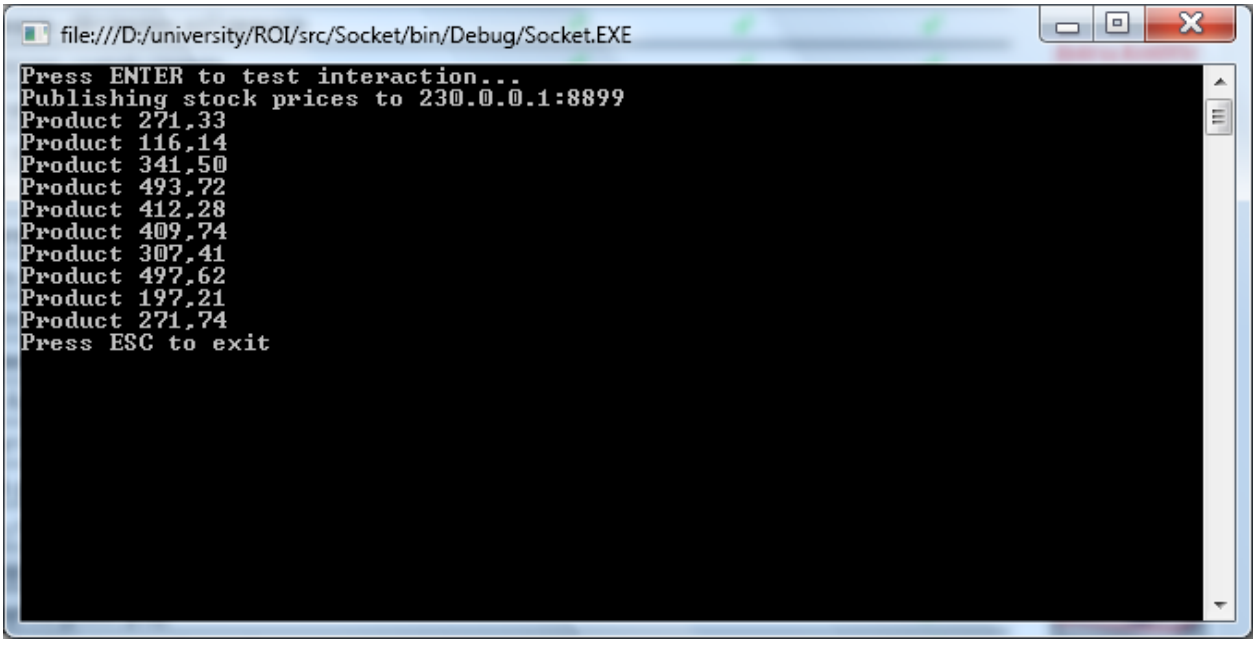



Рис.73. Результат роботи програми

### 10.6. Завдання для самостійної підготовки

1. До наведених варіантів асинхронних серверів написати варіанти асинхронних клієнтів.

2. Здійснити серверну обробку потоку сирих даних (потоку байтів), що поступають від клієнтів зі зберіганням їх в базі даних. Наприклад, існує множина пристроїв, кожен з яких раз в мінуту передає потік посекундних двобайтних значень певних характеристик. Організувати паралельну обробку. Обробити варіанти втрати зв'язку, невиходу в означений термін (здійснення прийому інформації з попередніх сеансів).

3. Реалізувати версію системи публікації-подписки цін на товари. Декілька серверів публікують свої ціни на товари, клієнти отримують ці ціни, виконують порівняння та виводять в GUI-інтерфейс користувача порівняння по вказаному товару. Здійснити прийом (зберігання та відображення) цін з інтервалом, визначеним користувачем.

4. Написати простішу програму (чат) на базі архітектури «клієнт-сервер» з використанням вивчених технологій.

# ПЕРЕЛІК ЛІТЕРАТУРНИХ ДЖЕРЕЛ

**[Aalst, 2002]** Wil van der Aalst, K.M. van HeeM. Workflow Management: Models, Methods, and Systems. IT press, Cambridge, MA, 2002. – 384 p.

**[Armbrust, 2010]** Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, Matei Zaharia: A view of cloud computing. Commun. ACM 53(4):50-58 (2010).

**[Bernstein, 2009]** Bernstein P.A. Principles of transaction processing. Eric Newcomer. - 2nd ed., 2009.- 378 p.

**[Bolton, 2010]** Christian Bolton, Justin Langford, Brent Ozar, James Rowland-Jones, Steven Wort. Professional SQL Server® 2008. Internals and Troubleshooting. Wiley Publishing, Inc., 2010.- 581 p.

**[Bool, 1854]** Boole, George (1854/1958). An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Proba-?ilities. Macmillan Publishers, 1854. Reprinted with corrections, Dover Publicat,-ons, New York, NY, 1958.

**[Brewer, 2000]** Eric A. Brewer. Towards robust distributed systems. (Invited Talk) Principles of Distributed Computing, Portland, Oregon, July 2000. http://www.cs.berkeley.edu/ ~ brewer/cs262b-2004/PODC-keynote.pdf

**[Brocke, 2010]** Jan vom Brocke, Michael Rosemann. Handbook on Business Process Management 1: Introduction, Methods, and Information Systems (International Handbooks on Information Systems). Springer-Verlag Berlin Heidelberg 2010. 636 p.

**[Cerami, 2002]** Ethan Cerami. Web Services Essentials. Distributed Applications with XML-RPC, SOAP, UDDI & WSDL. Publisher: O'Reilly, 2002. – 304 p.

**[CGI]** CGI Programming 101: Learn CGI Today! Chapter 4: Processing Forms and Sending Mail. – http://www.cgi101.com/book/ch4/text.html

**[CGI2]** Creating a C#/.NET CGI Executable. – http://www.west-wind.com/ WebLog/ posts/1143.aspx

**[Delaney, 2009]** K.Delaney, P.S.Randel. Microsoft® SQL Server® 2008 Internals. Microsoft Press; 1 edition (March 11, 2009) –784 p.

**[Deming, 1986]** Deming, W. Edwards. Out of the Crisis. MIT Press, 1986.- 507 p.

**[EBXML]** OASIS ebXML Message Specification, https://www.oasis-open.org/committees/ebxml-msg/documents/ebMS_v2_0.pdf

**[Foster, 2001]** Ian Foster, Carl Kesselman, The Grid 2, Second Edition: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishing 2003.- 748 p.

**[Gartner, 2008]** Gartner Says Worldwide IT Spending On Pace to Surpass $3.4 Trillion in 2008, http://www.gartner.com/it/page.jsp?id=742913

**[Gartner, 2010]** 2010 CIO Survey: A Time of Great IT Transition. http://www.itbusinessedge.com/ cm/ blogs/vizard/ gartner-2010-cio-survey-a-time-of-great-it-transition/?cs=38795

**[Gonzalez, 2009]** Gonzalez, Luis Miguel Vaquero etc: A break in the clouds: towards a cloud definition. Computer Communication Review. 39-1 (50-55) (2009).

**[Haas, 2007]** P. Haas. Semantic Technologies for Distributed Information Systems. Kit Scientific Publishing (31. Januar 2007). - 226 p.

**[Hammer, 1990]** Hammer M. Reengineering work: Don't automate, obliterate. Harv Bus Rev 68, 1990: pp.104–112.

**[Heimbigner, 1985]** HEIMBIGNER, D. AND MCLEOD, D. A federated architecture for information management. ACM Transactions on Information Systems (TOIS). Volume 3 Issue 3, July 1985. Pages 253-278.

**[HTTPR]** HTTPR Specification. http://www.ibm.com/developerworks/library/ws-httprspec/

**[Hyacinth, 1996]** Hyacinth S. Nwana. Software Agents: An Overview. Knowledge Engineering Review, Vol. 11, No 3, pp.1-40, Sept 1996. © Cambridge University Press, 1996. http://agents.umbc.edu/introduction/ao/

**[IIS]** Установка IIS. – http://technet.microsoft.com/ru-ru/library/ cc782498(WS.10) .aspx

**[Jia, 2005]** Weijia Jia, Wanlei Zhou. Distributed Network Systems: From Concepts to Implementations, Springer, 2005. – 513 p.

**[JMS]** Sun Microsystems. Java Message Service API Specification v1.1. Sun Microsystems, April 2002. http://www.oracle.com/technetwork/java/jms/ index.html

**[Kaplan, 2007]** Jeffrey M. Kaplan. Saas : Friend or foe? In Business Communications Review, pages 48-53, June 2007, http://smr-knowledge.com/wp-content/uploads/2010/01/EOS-SaaS-White-Paper-2008.pdf

**[Kelby, 2009]** J. Kelbey. Windows Server® 2008 Hyper-V™. Insider's Guide to Microsoft's Hypervisor, Wiley publishing inc., 2009. - 387 p.

**[Khan]** Rashid N. Khan . BPM на практике : жизненный цикл бизнес-процесса , http://bpms.ru/library/articles/bpm-lifecycle/index.html

**[Krowczyk, 2005]** Эндрю Кровчик, Винод Кумар и др. .Net. Сетевое программирование для профессионалов. Издательство «Лори». 2005. – 400 с.

**[Langley, 2009]** Langley, G. Moen, R., Nolan, K., Nolan, T., Norman, C., Provost, L. The Improvement Guide, 2nd Edition. Jossey-Bass, San Francisco.2009.- 512 p.

**[Liebig, 2001]** C. Liebig, S. Tai. Middleware - Mediated Transactions. In Proc. 3rd International Symposium on Distributed Objects and Applications (DOA 2001), IEEE, 2001. http://www.cin.ufpe.br/~redis/middleware/liebig-transactional01.pdf

**[Linthicum, 2009]** David S. Linthicum. Cloud Computing and SOA Convergence in Your Enterprise: A Step-by-Step Guide. Addison-Wesley Professional; 1 edition (September 29, 2009). - 264 p.

**[Martin, 1996]** David L. Martin, Adam Cheyer, and Gowang-Lo Lee, "Agent Development Tools for the Open Agent Achitecture". In Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM96). The Practical Application Company Ltd., Blackpool, Lancashire, UK, April 1996, pp. 387-404.

**[MSMQ]** Microsoft. Microsoft Message Queuing (MSMQ). http://msdn.microsoft.com/en-s/library/windows/desktop/ms711472(v=vs.85).aspx

**[MSOnline]** http://www.microsoft.com/online/

**[NDR]** CAE specification.Transfer Syntax NDR. – http://pubs.opengroup.org/onlinepubs /9629399/chap14.htm.

**[NIST, 2011]** Peter Mell, Timothy Grance. The NIST Definition of Cloud Computing. Recommendations of the National Institute of Standards and Technology. NIST Special Publication 800-145, Computer Security Division Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899-8930 September 2011. http://csrc.nist.gov/ publications/nistpubs/800-145/SP800-145.pdf

**[Parkhill, 1966]** Douglas F. Parkhill. The Challenge of the Computer Utility. Addison-wesley; 1st edition (January 1, 1966).- 207 p.

**[Peregudov, 1989]** Перегудов Ф.И., Тарасенко Ф.П. Введение в системный анализ. - М.: Высшая школа, 1989, с.320.

**[Pietzuch, 2003]** Pietzuch, P. R., Shand, B., and Bacon, J. (2003) A Framework for Event Composition in Distributed Systems. Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware 2003), Rio de Janeiro, Brazil; Springer-Verlag, Heidelberg, Germany. http://www.doc.ic.ac.uk/~prp/manager/doc/mw2003_ced_paper.pdf

**[Ray, 2009]** Chanda Ray. Distributed Database Systems. Pearson Education India, 2009. - 324 p.

**[Redkar, 2011]** Tejaswi Redkar. Windows Azure Platform Apress; 2 edition (December 7, 2011). - 602 p.

**[REST]** REST Web Services in ASP.NET 2.0 (C#). – http://www.codeproject.com/KB/aspnet/ RestServicesInASPNET2.aspx

**[REST2]** Build ReST based Web Services in .NET/C#. – http://www.codeproject. com/KB/webservices/ReSTBasedServiceForCSharp.aspx

**[Saltor, 1996]** Saltor, F., Campderrich, B. etc.: On Schema Levels for Federated DB Systems. In Yetongnon & Hariri (eds.): Proc. of the ISCA Int'l. Conf. on Parallel and Distributed Computing Systems (Dijon, 1996), ISCA, pp. 766-771.

**[Samos, 1998]** José Samos, Fèlix Saltor, Jaume Sistac, Agusti Bardés. Database architecture for data warehousing: An evolutionary approach. Database and Expert Systems Applications. Lecture Notes in Computer Science Volume 1460, 1998, pp 746-756.

**[Sheth, 1990]** Amit P. Sheth, James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. ACM Computing Surveys (CSUR) - Special issue on heterogeneous databases. Volume 22 Issue 3, Sept. 199, pp. 183 – 236.

**[Shewhart, 1986]** Shewhart W. Statistical method from the viewpoint of quality control. Dover Publications, NY. 1986. - 176 p.

**[Silver, 1995]** Mark S. Silver, M. Lynne Markus, Cynthia Mathis Beath (1995) The Information Technology Interaction Model: A Foundation for the MBA Core Course, MIS Quarterly, Vol. 19, No. 3, Special Issue on IS Curricula and Pedagogy (Sep., 1995), pp. 361-390.

**[SOAP]** Специфікація SOAP. http://www.w3.org/TR/soap/.

**[Tai, 2003]** Tai, S., Totok, A., Mikalsen, T., et al. (2003) Message Queuing Patterns for Middleware-Mediated Transactions. Proceedings of the SEM 2002, Orlando, FL; Springer-Verlag. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.10.3622

**[Tanenbaum, 2006]** Andrew Tanenbaum. Distributed systems: principles and paradigms, 2ed. 2006.

**[TIBCO]** TIBCO. TIBCO Rendezvous. http://www.tibco.com/products/ automation/messaging/low-latency/rendezvous/default.jsp

**[Tidwell, 2001]** Doug Tidwell. Programming Web Services with SOAP. O'Reilly Media, 2001 – 264 p.

**[WS-ReliableMessaging]** Web Services Reliable Messaging Protocol (WS-ReliableMessaging). February 2005. http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf

**[WSReliability]** Web Services Reliable Messaging. http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf

**[XML-RPC]** Офіційний сайт XML-RPC. http://www.xml-rpc.net/.

Наукове видання

Литвинов Олександр Анатолійович

Хандецький Володимир Сергійович

# РОЗПОДІЛЕНА ОБРОБКА ІНФОРМАЦІЇ

Монографія

Підписано до друку 26.04.13 Формат 70х100/8.

Умовн. друк. арк.. 36, 5

Тираж 300 прим. Зам. №1824.

Видавництво «**Акцент ПП**»

Пр.. Кірова, 91, м.Дніпропетровськ, 49054

Тел.. (056) 794-61-04(05)

Свідотство суб'єкта видавничої справи

Серія ДК №4122 від 27.07.2011.

Віддруковано в ТОВ «**Баланс-Клуб**»

пров..Верстатобудівельний, 4,

м. Дніпропетровськ, 49047

Тел.. (056) 370-44-25

Свідотство суб'єкта видавничої справи

Серія ДК №2119 від 03.03.2005.

**ISBN 978-966-494-025-9**