УДК 004.62 https://doi.org/10.15407/pp2021.01.036

Ю.А. Дивак

АНАЛІТИЧНИЙ ОГЛЯД ПІДХОДІВ ДО ІНТЕГРАЦІЇ ПРОГРАМНИХ СИСТЕМ

Виконаний порівняльний аналіз існуючих підходів до інтеграції програмних систем, з метою подальшого розроблення нових альтернативних підходів для вирішення задач інтеграції і композиції веб сервісів. Проведено інтеграцію платіжної системи в Інтернет-магазин одним із зазначених методів для більш глибокого розуміння проблем які виникають при інтеграції програмних систем. Розглянуті виклики, які стоять перед розробниками під час інтеграцій систем, та методи вирішення поставлених задач. Визначено ключові моменти на які слід звернути уваги про інтеграції програмних систем і взаємодії між ними. Об'єкт вивчення потребує нового погляду на проблему і нових альтернативних підходів, які принесуть більшу гнучкість, можуть підвищити продуктивність, знайти своє застосування у сфері інтеграції програмних систем і композиції веб-сервісів.

Ключові слова: Інтеграція програмних систем, композиція веб-сервісів.

Вступ

Великі компанії мають екосистеми, які включають у себе більше ніж одну програмну систему для підтримки бізнес процесів [1]. Наявність декількох систем працюючих разом дає багато переваг по відношенню до відокремлених незалежних монолітних систем: вони можуть працювати узгоджено розподіляючи ресурси, мати більш широкий функціонал, працювати незалежно та об'єднувати результати роботи. Тому, в корпоративному середовищі в сфері інформаційних технологій існує проблема інтеграції декількох відокремлених систем. Основне питання цієї проблеми, це як змусити відокремлені системи працювати разом та забезпечити об'єднаний набір функцій? Деякі програми можуть розроблятися власноруч вдома або в гаражі, тоді як інші купуються у сторонніх постачальників. Додатки працюють на декількох комп'ютерах, які можуть представляти кілька платформ і можуть бути географічно розподіленими. Деякі програми можуть працювати поза межами підприємства зі сторони партнерів або замовників. Доводиться інтегрувати програми які не були розроблені для інтеграції та взаємодії з іншими системами, та не можуть бути змінені, що ускладнює завдання. Ці проблеми та інші подібні до них ускладнюють інтеграцію додатків і взаємодію між системами.

Метою аналітичного огляду є розгляд існуючих підходів до інтеграції, для більш глибокого аналізу проблематики та розуміння потреб тих кого ця проблема стосується – це розробники корпоративних застосунків та інтеграційних рішень, корпорації метою яких є інтеграція з партнерами та колаборація з іншими учасниками ринку для досягнення економічних показників і конкурентоспроможності, наукове товариство яке зацікавлене в розробці нових підходів до існуючих проблем та їх вирішення. Ми розглянемо декілька підходів та визначимо позитивні та негативні сторони кожного з них.

Критерії оцінки та визначення підходу для інтеграції програмних систем

Інтеграція декількох систем потребує завжди різний набір властивостей і функцій, тому підходів з інтеграції існує декілька і кожен підхід задовольняє різні її потреби. Однак, як і будь-які складні технологічні зусилля, інтеграція додатків включає цілий ряд міркувань та наслідків, які слід враховувати під час вибору підходу до інтеграції.

Критерії інтеграції мають бути враховані під час вибору та розробки інтеграційного підходу. Який підхід інтеграції

корпоративних застосунків більш відповідає вказаним критеріям, такий і треба застосувати в конкретному випадку.

Перший критерій – це сама інтеграція програми. Чи потрібна інтеграція взагалі? Якщо можливо розробити окремий додаток, який не потребує взаємодії з будь-якими іншими програмами, то можемо повністю уникнути всіх проблем пов'язаних з інтеграцією застосунків. Однак, навіть у простого підприємства є декілька програмних продуктів, які повинні працювати разом, щоб забезпечити взаємодію для працівників підприємства, партнерів та клієнтів [3].

Зв'язність застосунків. Всі застосунки які приймають участь в інтеграції повинні мінімізувати свою залежність один від одного, щоб кожен міг розвиватися незалежно, не створюючи проблем для інших. Тісно зв'язані програми роблять численні припущення про те, як працюють інші програми (зазвичай на основі інтерфейсів, контрактів). Але коли програми змінюють і порушують ці припущення (інтерфейси, контракти), інтеграція порушується. Інтерфейс для інтеграції програм повинен бути достатньо конкретним для реалізації корисної функціональності, але достатньо загальним, щоб дозволити цій реалізації змінюватися за необхідності [3].

Простота інтеграції. При інтеграції програми в корпоративному середовищі розробники повинні мінімізувати зміни в програмі та мінімізувати необхідну кількість коду для інтеграції. Вирішення повинно бути простим і зрозумілим. Проте, як правило, необхідні зміни та новий код, щоб забезпечити хорошу функціональність інтеграції, і підходи з найменшим впливом на програму можуть не забезпечити найкращої інтеграції на підприємстві. Інтеграція повинна також забезпечувати гнучкість, що не завжди забезпечуються простими рішеннями [3].

Технологія інтеграції. Різні методи інтеграції вимагають різного обсягу спеціалізованого програмного та апаратного забезпечення. Ці спеціальні інструменти можуть бути дорогими і призвести до блокування постачальників (сервісів) та збільшити навантаження на розробників, які повинні розуміти як використовувати інструменти для інтеграції [3].

Формат даних. Інтегровані програми повинні узгоджувати формат даних, якими вони обмінюються, або повинні мати проміжний транслятор для уніфікації програм, які використовують різні формати даних. З часом формат даних може еволюціонувати, змінюватись та розширюватись, що може вплинути на функціональність програми. При чому узгодження повинно відбуватися як на загальному рівні (XML, JSON, BLOB), так і на рівні структур даних, які передаються між системами [3].

Своєчасність даних. Інтеграція повинна мінімізувати проміжок часу між програмами, коли одні відправляють дані, а інші отримують і можуть їх використовувати. Даними слід часто обмінюватися невеликими порціями, а не чекати обміну великим набором непов'язаних елементів, але канал передачі даних є вузьким місцем будь-якої системи, тому іноді краще зменшити кількість запитів між інтегрованими системами та відправляти дані одним набором, що підвищить швидкість взаємодії між системами. Інтегровані програми повинні бути проінформовані, як тільки спільні дані будуть готові до споживання. Затримка обміну даними повинна враховуватися в інтеграційному дизайні; чим довше триває доступ, тим більша вірогідність що дані застаріли і тим складнішою стає інтеграція [3].

Дані або функціональні можливості. Інтегровані програми можуть не обмінюватися даними, а використовувати функції, щоб кожна програма могла використовувати функціональність інших програм. Викликати віддалену функцію важко, і хоча це може здатися таким самим, як виклик локальної функції, воно працює зовсім по-іншому, що суттєво впливає на ефективність роботи інтеграції.

Прикладом такої інтеграції між двома відокремленими програмами, застосунком та базою даних, які можуть бути розташовані на різних серверах є можливість виклику віддалених процедур у базах даних Oracle та MySQL [3].

Асинхронність. Комп'ютерна обробка, як правило, синхронна, це значить що процедура чекає, поки виконується її підпроцедура, але можливий варіант виконання процедури асинхронно коли процедура не буде чекати закінчення виконання підпроцедури, а виконає її у фоновому режимі. Це особливо стосується інтегрованих програм, де віддалена програма може не працювати, або мережа може бути недоступною, у такому випадку вихідна програма може просто зробити доступними спільні дані або записати запит на виклик підпроцедури, але потім перейти до іншої роботи впевнена, що віддалений виклик буде викликано пізніше [3].

Існуючі підходи до інтеграції корпоративних застосунків

Представлено декілька підходів для інтеграції застосунків. Кожен з них у більшій чи в меншій мірі відповідає критеріям інтеграції. Як стверджено G. Hohpe та B. Woolf [3] інтеграція програмних систем може бути виконана чотирма підходами які називаються: Передача файлів, Спільна база даних, Виклик віддаленої процедури, Обмін повідомленнями.

1. Передача файлів (by Martin Fowler). Кожна програма виробляє файли з даними якими вона ділиться з іншими застосунками. Та отримує файли з даними, які були вироблені іншими застосунками.

Уявляючи, як організація працює з єдиного цілісного програмного забезпечення, розробленого з самого початку, щоб працювати в єдиній і цілісній формі. Звичайно, навіть найменші операції не працюють так. Кілька програмних продуктів обробляють різні аспекти діяльності підприємства. Це пов'язано з низкою причин.

– Корпорації купують програмні системи, розроблені сторонніми організаціями.

– Різні системи будуються в різний час, що призводить до різного вибору технологій.

– Будь-які системи будуються особами, досвід та уподобання яких приводить до різних підходів до побудови додатків.

Розроблення програмного продукту та надання цінності компанії та бізнесу є більш важливим, ніж забезпечення вирішення питань інтеграції з іншими системами, особливо коли ця інтеграція не додає ніякої цінності програмі, що розробляється. Як результат, будь-яка організація повинна турбуватися про обмін інформацією між різними програмами. Вони можуть бути написані різними мовами, на основі різних платформ і з різними припущеннями про те, як працює бізнес. Зв'язування таких додатків вимагає великих знань про розуміння того, як пов'язати програми як на бізнес-рівні, так і на технічному рівні. Щоб налагодити взаємодію, треба мінімізувати те, що потрібно знати програмі про те, як працюють інші програми. Потрібен загальний механізм передачі даних, який може використовуватися різними мовами та платформами. Для цього потрібно мати мінімальну кількість спеціалізованого обладнання та програмного забезпечення, використовуючи те, що є у наявності підприємства. Файли – це універсальний механізм зберігання, вбудований в будь-яку корпоративну операційну систему, доступний на будь-якій мові підприємства. Найпростішим підходом було б якось інтегрувати програми за допомогою файлів (рис. 1).

Нехай кожна програма створює файли, що містять інформацію, яку повинні використовувати інші програми. Інтегратори несуть відповідальність за перетворення файлів у різні формати.

Важливим рішенням щодо файлів є формат, який використовувати. Дуже рідко результат однієї програми буде саме

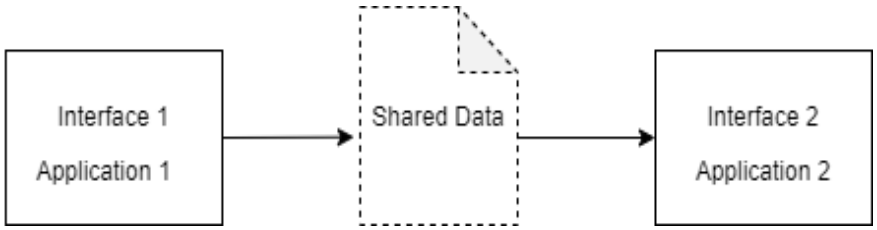

Рис. 1. Інтеграція через Передачу файлів

тим, що потрібно для іншої, тож доведеться досить дорого обробляти та перетворювати файли. Всі програми, які використовують файл, повинні його прочитати, та вміти використовувати на ньому інструменти обробки. Як результат, із часом стандартні формати файлів виросли. Системи мейнфреймів зазвичай використовують канали даних на основі форматів файлових систем COBOL. Системи Unix використовують текстові файли. Сучасність полягає у використанні XML та JSON. Навколо кожного з цих форматів склалася галузь читачів файлів, письменників файлів та засобів трансформації. Інше питання, пов'язане з файлами, – це коли їх створювати та споживати. Оскільки для створення та обробки файлу потрібні певні зусилля, але не має бажання надто часто працювати з ними. Зазвичай є якийсь регулярний діловий цикл, який визначає рішення: щоночі, щотижня, щокварталу тощо. Програми пристосовують до конкретно часу в яких інформація у файлі доступна та є актуальною, і обробляють його у свій час. Великою перевагою файлів є те, що інтегратори не потребують знання внутрішніх елементів програми. Команда заявок зазвичай надає файл. Вміст та формат файлу узгоджуються з інтеграторами. В результаті різні додатки які приймають участь в інтеграції досить добре відокремлені один від одного. Кожна програма може вільно вносити внутрішні зміни, не впливаючи на інші програми, за умови, що вони все одно створюють однакові дані у файлах у тому ж форматі. Файли фактично стають інтерфейсом кожного додатка. Частина того, що спрощує передачу файлів, полягає у тому, що не потрібні додаткові інструменти чи пакети інтеграції, але це також означає, що розробники повинні зробити багато роботи самостійно. Програми повинні узгодити правила іменування файлів та каталоги, в яких вони відображаються. Автор файлу повинен реалізувати стратегію, щоб зберегти імена файлів унікальними. Програми повинні узгодити, як будуть видалятися, і визначатися старі файли. Додатки повинні впровадити механізм блокування або дотримуватися норми синхронізації, щоб гарантувати, що одна програма не намагається прочитати файл, а інша все ще пише його. Якщо всі програми не мають доступу до одного диска, тоді деяка програма повинна взяти на себе відповідальність за передачу файлу з одного диска на інший. Однією з найбільш очевидних проблем передачі файлів є те, що оновлення, як правило, зустрічаються нечасто, в результаті чого системи можуть вийти з синхронізації. Система керування клієнтами може обробляти зміну адреси та видавати файл витягу щовечора, але система виставлення рахунків може відправити рахунок на стару адресу того ж дня. Іноді відсутність синхронізації не є великою проблемою. Часто очікується певне відставання в отриманні інформації, навіть від комп'ютерів. В інших випадках використання несвіжої інформації є катастрофою для роботи програм. Вирішуючи, коли створювати файли, потрібно враховувати потреби споживачів у актуальності даних. Якщо клієнт того самого дня змінює свою адресу за допомогою двох різних систем, але одна з них помиляється і отримує неправильну назву вулиці, то буде дві невідповідні адреси клієнта. Тоді знадобиться якийсь спосіб на вирішення цього питання. Чим довший період між передачею файлів, тим більш імовірною і нестерпнішою стане ця проблема. Звичайно,

файли можна створювати частіше для підтримки актуальності даних. Інша проблема полягає в керуванні всіма створеними файлами. Повний перелік можливостей підходу основаного на файловій системі набагато ширший, але при обробці файлу витрачається багато ресурсів, що є надмірним, якщо хочемо швидко створити багато файлів [3].

2. Спільна база даних. Застосунки зберігають дані якими вони хочуть ділитися в спільній базі даних.

Для сучасного бізнесу хочемо, щоб усі мали якомога більше актуальних даних. Це не просто зменшення помилок, а збільшення довіри до даних. Швидке оновлення також дозволяє краще усунути невідповідності. Чим частіше синхронізуємось, тим менше шансів отримати невідповідності і тим менше зусиль витрачаємо на вирішення. Але якими б швидкими не були зміни, все одно будуть проблеми.

Існує багато прикладів семантичного дисонансу при застосуванні спільної бази даних, які набагато складніше розглядати, ніж суперечливі формати даних. (Для набагато глибшого обговорення цих питань варто прочитати Data and reality, a timeless perspective on perceiving and managing information in our imprecise world, William Kent [4]).

Якщо потрібна централізована, узгоджена база даних, до якої мають спільний доступ усі додатки, щоб кожен із них мав доступ до будь-якої спільної інформації, і не було б за потрібне інтегрувати програми, через збереження своїх даних в одній спільній базі даних (рис. 2).

Якщо сімейство інтегрованих додатків покладається на одну і ту ж базу даних, то можемо бути впевнені, що вони завжди постійно узгоджуються. Якщо отримуємо одночасне оновлення окремого фрагмента даних з різних джерел, щоб уникнути колізій для цього існують системи керування транзакціями, які керуються принципами ACID [5]. Використання спільної бази даних значно полегшується завдяки широкому розповсюдженню реляційних баз даних на базі SQL. Практично всі платформи для розробки додатків можуть працювати з SQL, часто з досить складними інструментами. Тому не доведеться турбуватися про декілька форматів файлів. Оскільки будь-яка програма в будь-якому випадку повинна використовувати SQL, це уникає додавання іншої технології, якою кожен може володіти. Оскільки всі використовують одну і ту ж базу даних, це вирішує проблеми семантичного дисонансу [3].

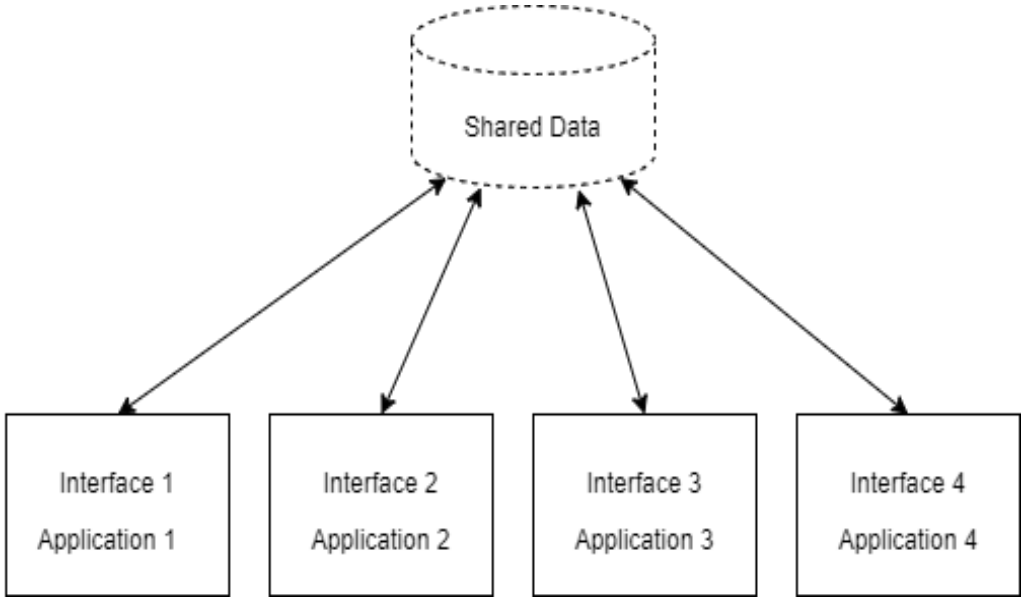

Рис. 2. Інтеграція через Спільну базу даних

Однією з найбільших труднощів спільної бази даних є створення відповідного дизайну (мається на увазі архітектура системи). Інтеграційна база даних повинна мати схему, яка задовольнить потреби всіх клієнтських систем, ця схема завжди більш загальна і більш складна. База даних розробляється декількома організаціями тому змінити щось у базі складно, через те, що зміни мають погоджуватися між клієнтськими застосунками. Розробка уніфікованої схеми, яка може задовольнити потреби багатьох додатків, є дуже складною справою, що часто призводить до створення схеми, з якою програмістам важко працювати. Якщо критична програма, ймовірно, зазнає затримок для роботи з уніфікованою схемою, то часто виникає непереборний тиск для відокремлення. Конфлікти між відділами часто посилюють цю проблему. Іншим, більш жорстким обмеженням для спільної бази даних є зовнішні програмні системи. Частіше вони не працюватимуть з іншою схемою, окрім власної. Навіть якщо є якийсь простір для адаптації, це, швидше за все, буде набагато обмеженішим, ніж хотіли б інтегратори. Ця проблема також поширюється на інтеграцію після розробки. Навіть якщо впорядкувати всі свої програми, все одно виникне проблема інтеграції, якщо відбудеться злиття компаній. Кілька програм, які використовують Спільну базу даних для частого читання та модифікації одних і тих самих даних, можуть спричинити вузькі місця у роботі та навіть тупикові ситуації, оскільки кожна програма блокує інші дані. Коли програми розподіляються між кількома комп'ютерами, база даних повинна бути розподілена також, щоб кожна програма могла отримати доступ до бази даних локально, що заплутує проблему, на якому комп'ютері слід зберігати дані. Розподілена база даних із конфліктами блокування може легко стати перешкодою продуктивності [3].

Існує два підходи до застосування баз даних. База даних може бути або прив'язаною до конкретної системи і належати їй, або вона може бути інтеграційною, і зберігати спільні дані з декількох систем. Різниця між ними полягає у тому хто контролює потік даних. В останні часи розвивається сервісно-орієнтований підхід до розробки корпоративних програмних систем з власною базою даних, які взаємодіють через сервісні інтерфейси, ефективно замінюючи інтеграцію через спільну базу даних інтеграцією на основі виклику віддаленої процедури або обміном повідомлень.

Плюс інтеграційної бази даних – це інтеграція, яка не потребує відокремленого шару інтеграційних сервісів у застосунку. Всі зміни в базі даних стають доступними для всіх клієнтських застосунків одночасно, що робить спільні дані більш синхронізованими.

З іншого боку спільна база даних веде до значних проблем тому, що вона стає точкою зв'язності застосунків. Зазвичай це дуже глибока зв'язність, яка збільшує ризик пов'язаний із змінами системи та її розвитком. Більшість спеціалістів вважає, що треба уникати цього підходу для інтеграції.

3. Виклик віддалених процедур (by Martin Fowler) – кожен застосунок має процедури, які можна викликати віддалено і програми таким викликом запускають процес обміну даними.

Парадигма RPC була використана для впровадження багатьох повсякденних систем. Від програм нижчого рівня, таких як Мережеві файлові системи [6] та віддаленого прямого доступу до пам'яті [7], до протоколів доступу до розвитку екосистеми мікросервісів, RPC використовується скрізь. RPC має різноманітні програми – SunNFS [6], Twitter Finagle (Eriksen, 2013), Apache Thrift (Prunicki, 2009), Java RMI [8], SOAP, CORBA (Group, 1991) та gRPC від Google (Google, n. d.).

Існує декілька підходів до віддаленого виклику процедур (RPC): CORBA, COM, .NET Remoting, Java RMI і т. д. Вони відрізняються за кількістю підтримуваних систем та простотою використання. Часто такі середовища мають додаткові можливості, такі як транзакції.

Передача файлів та спільна база даних дозволяють програмам обмінюватися своїми даними, що є важливою частиною інтеграції програм, але просто обміну даними часто буває недостатньо. Часто зміни даних призводять до того, що доводиться робити зміни в різних додатках. Наприклад, зміна адреси може бути простою зміною даних, або це може спричинити реєстрацію та юридичні процеси з урахуванням різних правил у різних юридичних юрисдикціях. Наявність однієї програми для виклику таких процесів у іншій вимагало б від програм занадто багато знання про внутрішні функції інших програм. Ця проблема відображає класичні проблеми у розробці додатків. Одним з найпотужніших механізмів структурування при розробці додатків є механізм інкапсуляції – де модулі приховують свої дані через інтерфейс виклику функції. Таким чином, вони можуть перехоплювати зміни в даних, щоб виконувати різні дії, які їм потрібно робити, коли дані змінюються. Спільна база даних забезпечує велику, неінкапсульовану структуру даних, що значно ускладнює контроль даних. Передача файлів дозволяє програмі реагувати на зміни під час обробки файлу, але процес затримується. Той факт, що спільна база даних має некапсульовані дані, також ускладнює ведення сімейства інтегрованих додатків. Багато змін у будь-якій програмі можуть спричинити зміни в базі даних, які мають значний вплив на кожен додаток. Як результат, системи, які використовують спільну базу даних, часто дуже неохоче змінюють базу даних, а це означає, що робота з розробки додатків набагато менше реагує на зміни потреб бізнесу [3].

Необхідний механізм який дозволить одному застосунку викликати функції в інших застосунках, передавати дані, які потрібні для спільного використання та викликати функцію, яка інформує програму-отримувач як обробляти дані [3].

Розробка кожної програми – це величезний проект з інкапсульованими даними. Забезпечення інтерфейсу, який дозволяє іншим програмам взаємодіяти з цим застосунком [3].

Віддалений виклик процедур застосовує принципи інкапсуляції до інтеграції застосунків (рис. 3). Якщо програма потребує деякої інформації, яка належить іншій програмі, вона напряму посилає запит до застосунку, який володіє інформацією. Якщо одній програмі необхідно змінити дані іншої, вона посилає запит для зміни інформації. Кожний застосунок підтримує цілісність власних даних, а також може змінювати дані незалежно від інших застосунків.

На сьогодні лідерами у використанні є веб служби, які використовують такі стандарти як: SOAP і XML. Цінною особливістю є те, що вони легко працюють з HTTP, який проходить через брандмауери.

Той факт, що існують методи які обробляють дані, спрощує семантичний дисонанс. Застосунки можуть забезпечувати декілька інтерфейсів для одних і тих же даних, дозволяючи одним користувачам бачити один стиль, а іншим – інший. Оновлення можуть використовувати декілька інтерфейсів, що дає більшу можливість для підтримки декількох методів взаємодії. Однак інтеграторам не зручно

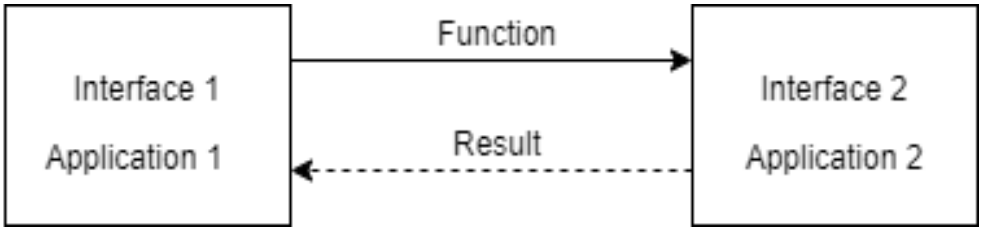

Рис. 3. Віддалений виклик процедур

додавати конвертери, тому важливо узгоджувати інтерфейси взаємодії з іншими застосунками.

Оскільки розробники звикли до виклику процедур, RPC добре співпрацює з тим до чого вони звикли. Насправді це скоріше недолік. Між віддаленими та локальними викликами існує велика різниця у швидкодії та надійності. Якщо розробники не розуміють як використовувати віддалені процедури правильно то це може призвести до розробки повільних та ненадійних систем [8].Інкапсуляція допомагає зменшити зв'язок програм, усуваючи велику спільну структуру даних, але програми ще досить тісно пов'язані між собою. Віддалені виклики, які підтримує кожна система, мають тенденцію пов'язувати різні системи у зростаючий вузол. Зокрема, послідовність – виконання певних дій у певному порядку, яка може ускладнити самостійну зміну систем. Це проблеми, які не важливі у межах однієї програми, а стають важливі при інтеграції декількох програм.

Спеціалісти, розробляючи інтеграцію як єдину програму, не підозрюють про правила зміни.

4. Відправлення повідомлень – кожна програма приєднана до спільної системи повідомлень, яка ділиться даними та запускає будь-які процедури використовуючи повідомлення (рис. 4).

Асинхронні повідомлення – це принципово прагматична реакція на проблеми розподілених систем. Надсилання повідомлення не вимагає одночасної роботи і готовності обох систем.

Існує багато прикладів застосування підходу передачі повідомлень до інтеграції застосунків через Enterprise Service Bus (ESB), це такі як Dell Boomi [9], Informatica [10], JitterBit [11], MuleSoft [12], Oracle [13], SnapLogic [14], as well Guarana [15].

Крім того, розгляд асинхронної комунікації змушує розробників визнати, що робота з віддаленим додатком відбувається повільніше, що заохочує розробку компонентів з високою згуртованістю (багато роботи локально) та низькою адгезією (вибіркова робота віддалено). Системи обміну повідомленнями також дозволяють роз'єднання, яке можливо отримати під час використання передачі файлів. Повідомлення можуть бути трансформовані під час передачі, але ні відправник, ні отримувач не знають про трансформацію. Дійсно, роз'єднання дозволяє інтеграторам транслювати повідомлення на кілька приймачів, підтримує вибір одного з багатьох потенційних приймачів та інші топології, які

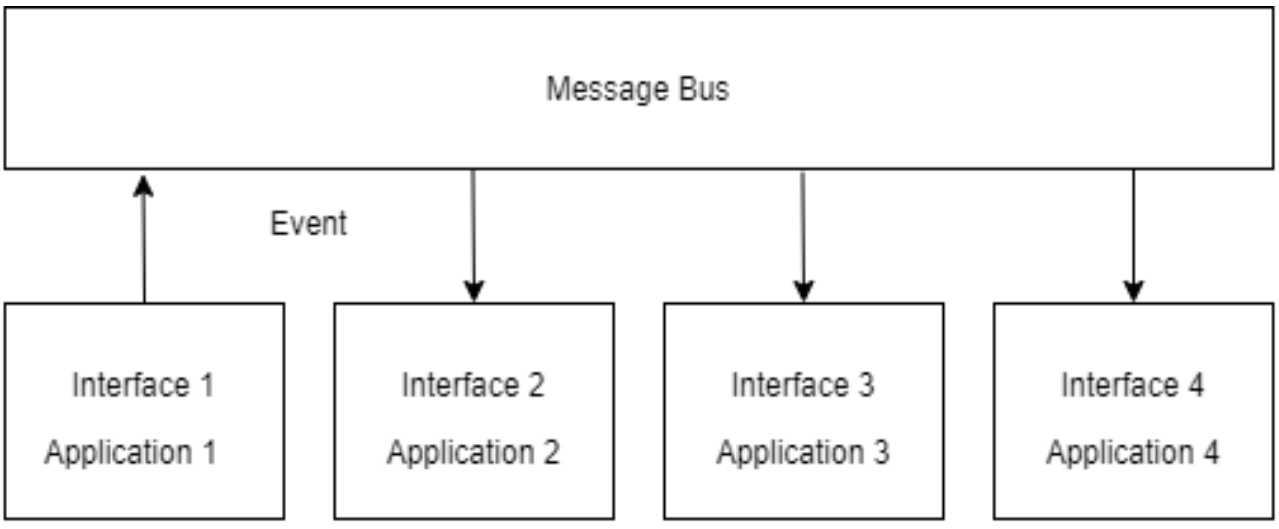

Рис. 4. Інтеграція через відправку повідомлень

дозволяють інтеграцію відокремлювати від розробки додатків. Оскільки дані проблеми, як правило, відокремлюють розробку додатків від інтеграції програм, цей підхід працює скоріше з людською природою, а не проти. Трансформація означає, що окремі програми можуть мати досить різні концептуальні моделі. Звичайно, це означає, що відбудеться семантичний дисонанс, але з точки зору обміну повідомленнями. Міра, яку вживає **Спільна база даних**, щоб уникнути семантичного дисонансу, занадто складна, щоб працювати на практиці, і не може бути виконана після об'єднання пакетів [3].

При надсиланні невеликих повідомлень, дозволяється програмам взаємодіяти між собою, а також обмінюватися даними. Можна запросити інформацію та швидко відповісти. Хоча така співпраця не буде такою швидкою, як віддалений виклик процедури, абоненту не потрібно зупинятися, поки обробляється повідомлення та повертається відповідь. І обмін повідомленнями не такий повільний, як думають багато людей – багато рішень для обміну повідомленнями зароджується у галузі фінансових послуг, де тисячі біржових котирувань або торгів повинні проходити через систему обміну повідомленнями щосекунди.

Висока частота повідомлень у програмі обміну повідомленнями зменшує багато проблем із невідповідністю, що спричиняє неприємну передачу файлів, але не повністю їх усуває. Виникатимуть проблеми із запізненням, оскільки системи не оновлюються одночасно.

Асинхронний дизайн – це не те, що вивчає більшість програмістів, і в результаті існує ціла низка різних правил і методів характерних вивченню такого дизайну. Тестування та налагодження є суттєво складнішим у середовищі обміну повідомленнями.

Можливість перетворення повідомлень має приємну перевагу, дозволяючи додаткам набагато відокремитись один від одного, ніж у віддаленому виклику процедур та передачі файлів. Але ця незалежність насправді означає, що інтеграторам часто залишається писати багато безладного коду, щоб все поєднати.

Вирішивши використовувати **Повідомлення** для системної інтеграції, існує низка нових питань, які слід розглянути і практики, які використовуватимемо. **Як передавати пакети даних?** Відправник надсилає дані повідомлення отримувачу через канал повідомлення, що з'єднує відправника та отримувача. **Як дізнатися куди надсилати дані?** Якщо відправник не знає, куди направити дані, він може надіслати дані маршрутизатору повідомлень, який направить дані до належного отримувача. **Як дізнатися який формат даних використовувати?** Якщо відправник та отримувач не погоджуються щодо формату даних, відправник може направити дані до **Перекладача** повідомлень, який перетворить дані у формат отримувача, а потім перенаправить дані отримувачу. Якщо ви розробник додатків, то виникає питання **як підключити програму до системи обміну повідомленнями?** Додаток, який бажає використовувати обмін повідомленнями, реалізує кінцеві точки повідомлень для фактичного надсилання та отримання.

Приклад інтеграції двох застосунків методом виклику віддалених процедур (RPC)

Для більш глибокого розуміння проблем, які виникають під час інтеграції застосунків розглянемо практичний приклад інтеграції додатків через віддалений виклик процедур. Нам потрібно інтегрувати два застосунки один з яких є сервісом, який надає постачальник сервісу, у нашому випадку постачальником сервісу є процесінговий центр, а сервіс це платіжна система, яка приймає платежі з банківських карт VISA та MASTERCARD та розподіляє платіж у будь-який банк утримуючи процент комісії.

Споживачем сервісу є Інтернет-магазин з продажу будь-чого. Задача ускладняється тим, що взаємодія застосунків повинна бути налагоджена з двох сторін – із сторони кінцевого користувача

(frontend), а також із сторони застосунку (backend). Кожна позитивна відповідь з платіжної системи буде реєструватися в системі інтернет-магазину для подальшої обробки замовлення та відслідковування коштів на рахунках (рис. 5).

Треба чітко розуміти, що ми маємо дві абсолютно незалежні системи у яких є фронтенд частина і бекенд частина які повинні взаємодіяти між собою.

Зі сторони кінцевого користувача інтеграція здійснюється за допомогою вбудованого айфрейму, якщо сервіс платіжної системи побудований правильно з можливістю адаптації візуальної частини під потреби споживача сервісу, то кінцевий користувач може не відрізнити вбудований сервіс і не зрозуміє, що водночас він працює з сервісом платіжної системи.

Проблема CORS (крос-доменних запитів) та зберігання cookies сесії вирішуються за допомогою реверс проксі, який можна налаштувати, наприклад, за допомогою веб серверу nginx.

Зі сторони серверної частини застосунку інтеграція проводиться за допомогою підходу виклику віддалених процедур через REST API.

Рис. 5. Інтеграція платіжної системи в Інтернет-магазин

Висновки

Передача файлів дозволяє програмам обмінюватися даними, але це може бути не своєчасною, а своєчасність може бути критичною проблемою в інтеграції. Якщо зміни не працюють швидко через декілька додатків, то, можливо допустити неправильні дії через застарілість даних. **Передача файлів** також може недостатньо застосовувати формат даних. Багато проблем при інтеграції виникають через несумісні способи обробки даних. Часто вони представляють тонкі бізнес-проблеми, які можуть мати величезний ефект. **Передача файлів** дозволяє програмі реагувати на зміни під час обробки файлу, тримати програми дуже добре відокремленими, але ціною своєчасності даних. Системи просто не встигають одна за одною. Поведінка співпраці занадто повільна.

Спільна база даних зберігає спільні дані, але ціною зв'язку всіх застосунків з базою даних. Цей підхід теж не справляється зі спільною поведінкою.

Передача файлів та **Спільна база** даних дозволяють програмам ділитися своїми даними, але не своїми функціональними можливостями.

Віддалений виклик процедур дозволяє програмам обмінюватися функціоналом, але тісно поєднує їх процеси. Часто завдання інтеграції полягає у тому, щоб зробити співпрацю між окремими системами якомога своєчасною, не поєднуючи системи між собою, таким чином робить їх ненадійними, як з точки зору виконання програми, так і розробки програми.

Зіткнувшись з цими проблемами, **віддалений виклик процедури** видається привабливим вибором. Але розширення моделі, яка використовується для одного додатка, при інтеграції додатків попадає на безліч інших слабких місць. Ці слабкі сторони починаються з основних проблем розподіленого розвитку. Незважаючи на те, що виклики віддалених процедур виглядають як місцеві виклики, вони не діють однаково. Віддалені виклики відбуваються повільніше і можуть не вдатися. Працюючи з кількома програмами, які обмінюються даними на одному підприємстві, не бажано, щоб збій однієї програми призвів до збою в усіх інші програмах. Крім того, не бажано розробляти систему, вважаючи, що виклики швидкі (але вони можуть бути досить повільними), і щоб кожна програма розпізнавала деталі інших програм, навіть якщо це лише деталі про їх інтерфейси.

Нам потрібно дещо на зразок **передачі файлів**, де багато невеликих пакетів даних можна швидко створити, легко передати, а програма-отримувач автоматично отримує повідомлення, коли новий пакет доступний для споживання. Передача потребує механізму повторної спроби, щоб переконатися, що вона успішна. Деталі будь-якої структури диска або бази даних для зберігання даних потрібно приховувати від програм, щоб, на відміну від **Спільної бази даних**, схема зберігання та деталі могли бути легко змінені, щоб відобразити зміни потреб підприємства. Одна програма повинна мати можливість надіслати пакет даних іншій програмі, щоб викликати поведінку в іншій програмі, наприклад, **віддалений виклик процедури**, але без схильності до відмови. Передача даних повинна бути асинхронною, щоб відправнику не потрібно було чекати на приймачі, особливо коли необхідна повторна спроба.

Проблема інтеграції програмних систем залишається відкритою. Крім існуючих підходів до інтеграції, предмет потребує нових альтернативних підходів до інтеграції, які відкриють нові шляхи для налагодження взаємодії, запропонують нові можливості та покриють потреби корпоративних програмних систем [2]. Інтеграція декількох систем завжди потребує різний набір властивостей і функцій (таблиця), тому системи відрізняються за обсягом, за підходами і технологіями. Інтеграція корпоративних застосунків не проста річ, яка потребує зважених технічних а також бізнес рішень.

Таблиця. Порівняння властивостей існуючих підходів

	Передача файлів	Спільна база даних	RPC	Повідомлення
Зв'язність	+	+	+	–
Швидкість	–	+	–	–
Інтеграція функціональності	–	–	+	+
Асинхронність	–	–	–	+
Можливість відмови	–	–	+	+
Гнучкість	+	–	+	+

Література

1. Rosa-Siqueira F.J., Basto-Fernandes V. Frantz R.Z. (2017) *Enterprise application integration: Approaches and platforms to design and implement solutions in the cloud.*
2. Soomro T.R., Awan A.H. (2012) Challenges and Future of Enterprise Application Integration, UAE, International Journal of Computer Applications.
3. Hohpe G., Woolf B. (2003) Enterprise integration patterns: Designing, building, and deploying messaging solutions, Addison-Wesley Professional.
4. Kent W. (2012) Data and reality, a timeless perspective on perceiving and managing information in our imprecise world. USA Technics Publications, LLC.
5. Kleppman M., (2017) Designing Data-Intensive Applications: The big ideas Behind Reliable, Scalable, and Maintainable Systems, USA, O'Reilly Media, Inc.
6. Sandberg R., Goldberg D., Kleiman S., Walsh D., & Lyon B. (1985). Design and implementation of the Sun network filesystem. *In Proceedings of the Summer*, USA CA, Sun Microsystems, Inc.
7. Kalia A., Kaminsky M., & Andersen D.G. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. *In 12th USENIX Symposium on Operating Systems Design and Implementation*, (OSDI 16: Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation) (P. 185–201).
8. Wollrath A., Riggs R., & Waldo J. (1996). A Distributed Object Model for the Java System, *Conference on Object-Oriented Technologies Toronto*, Ontario, Canada, June 1996.
9. Dell Boomi Home. https://boomi.com, 2017.
10. Informatica Home. https://www.informatica.com/products/cloud-integration, 2017.
11. Jitterbit Home. http://www.jitterbit.com, 2017
12. Mulesoft Home. https://www.mulesoft.com/, 2017.
13. Oracle Cloud Home. https://cloud.oracle.com/integration, 2017.
14. SnapLogic Home. https://www.snaplogic.com, 2017.
15. Frantz R.Z., Corchuelo R. (2016) On the design of a maintainable software development kit to implement integration solutions. *The Journal of Systems and Software, Journal of Systems and Software.* Vol. 111, January 2016. P. 89–104.

References

1. Rosa-Siqueira F.J., Basto-Fernandes V. Frantz R.Z. (2017) *Enterprise application integration: Approaches and platforms to design and implement solutions in the cloud.*
2. Soomro T.R., Awan A.H. (2012) Challenges and Future of Enterprise Application Integration, UAE, International Journal of Computer Applications.
3. Hohpe G., Woolf B. (2003) Enterprise integration patterns: Designing, building, and deploying messaging solutions, Addison-Wesley Professional.

4. Kent W. (2012) Data and reality, a timeless perspective on perceiving and managing information in our imprecise world. USA Technics Publications, LLC.
5. Kleppman M., (2017) Designing Data-Intensive Applications: The big ideas Behind Reliable, Scalable, and Maintainable Systems, USA, O'Reilly Media, Inc.
6. Sandberg R., Goldberg D., Kleiman S., Walsh D., & Lyon B. (1985). Design and implementation of the Sun network filesystem. *In Proceedings of the Summer*, USA CA, Sun Microsystems, Inc.
7. Kalia A., Kaminsky M., & Andersen D.G. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. *In 12th USENIX Symposium on Operating Systems Design and Implementation*, (OSDI 16: Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation) (P. 185–201).
8. Wollrath A., Riggs R., & Waldo J. (1996). A Distributed Object Model for the Java System, *Conference on Object-Oriented Technologies Toronto*, Ontario, Canada, June 1996.
9. Dell Boomi Home. https://boomi.com, 2017.
10. Informatica Home. https://www.informatica.com/products/cloud-integration, 2017.
11. Jitterbit Home. http://www.jitterbit.com, 2017.
12. Mulesoft Home. https://www.mulesoft.com/, 2017.
13. Oracle Cloud Home. https://cloud.oracle.com/integration, 2017.
14. SnapLogic Home. https://www.snaplogic.com, 2017.
15. Frantz R.Z., Corchuelo R. (2016) On the design of a maintainable software development kit to implement integration solutions. *The Journal of Systems and Software, Journal of Systems and Software.* Vol. 111, January 2016. P. 89–104.

Одержано 20.01.2021

Про автора:

Дивак Юрій Андрійович,
аспірант
Інституту програмних систем
Національної академії наук України.

Місце роботи автора:

Integration Full-stack software engineer,
Mediastream company.

E-mail: yurii.dyvak@ukma.edu.ua