БАКАЛАВРСЬКА РОБОТА

БР.ІП – 01.00.00.000 ПЗ

Група ІП-22-1

Когут Ольга

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Когут Ольга Василівна

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Методи забезпечення надійності розподілених програмних систем
(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____Когут О.В._____
(підпис, ініціали та прізвище здобувача)

Науковий керівник ____Ксенич Андрій Іванович, доцент, к.т.н. _____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____2026 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Когут Ользі Василівні

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Методи забезпечення надійності розподілених програмних систем"

керівник проекту (роботи) доцент, к.т.н. Ксенич А.І.

затверджені наказом вищого навчального закладу від **“** 21 ” квітня 2026 р. № 179/7

2. Строк подання студентом проекту (роботи) 10 червня 2026 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Аналіз вимог до програмного забезпечення
2. Аналіз вимог і постановка задачі
3. Проектування системи
4. Реалізація проекту
5. Тестування та впровадження

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1 Потік даних у стеку протоколів (рис. 1.1, ст. 14)
2 Великі повідомлення фрагментуються для передачі (рис. 1.2, ст. 16)
3 Збої системи (рис. 2.2, ст. 47)
4 Середовище C# .NET (рис. 3.1, ст. 63)
5 Процес на основі ланцюга Марков (рис. 3.4, ст. 70)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання **2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	17.03.2026	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	21.04.2026	виконано
3	Побудова моделі або алгоритму власного рішення	01.05.2026	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	21.05.2026	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	09.06.2026	виконано

Студент _____Когут О.В.
(підпис)

Керівник роботи _____Ксенич А.І.
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 90 сторінок, 12 рисунки, 1 таблицю, список використаних джерел із 40 найменувань.

Метою роботи є дослідження методів забезпечення надійності розподілених програмних систем та аналіз підходів до побудови відмовостійких, масштабованих і ефективних програмних рішень.

Об’єкт дослідження: розподілені програмні системи.

Предмет дослідження: методи, моделі та інструменти забезпечення надійності розподілених систем, включаючи механізми відмовостійкості.

Результати дослідження: проведено аналіз причин відмов у розподілених системах, досліджено методи виявлення та обробки збоїв, розглянуто сучасні підходи до побудови надійних систем, включаючи мікросервісну та serverless-архітектури.

У першому розділі розглянуто теоретичні основи забезпечення надійності розподілених систем, основні поняття, компоненти та принципи їх побудови, а також базові комунікаційні служби.

У другому розділі досліджено причини виникнення відмов, методи їх виявлення та діагностики, а також технології обміну даними.

У третьому розділі розглянуто практичні аспекти реалізації надійних розподілених систем, використання сучасних платформних технологій, багаторівневих архітектур та проведено аналіз ефективності сучасних підходів, зокрема serverless.

Висновок: застосування сучасних методів забезпечення надійності дозволяє підвищити стійкість розподілених систем до збоїв, забезпечити безперервність їх роботи та покращити ефективність обробки даних у складних інформаційних середовищах.

КЛЮЧОВІ СЛОВА: РОЗПОДІЛЕНІ СИСТЕМИ, НАДІЙНІСТЬ, ВІДМОВОСТІЙКІСТЬ, МІКРОСЕРВІСИ, SERVERLESS, ВИЯВЛЕННЯ ЗБОЇВ, ОБМІН ДАНИМИ, ХМАРНІ ОБЧИСЛЕННЯ.

ANNOTATION

The bachelor's thesis contains of 90 pages, 12 figures, 1 tables, and a reference list with 40 sources.

The aim of the work is to study methods for ensuring the reliability of distributed software systems and to analyze approaches to building fault-tolerant, scalable, and efficient software solutions.

Research object: distributed software systems.

Research subject: methods, models, and tools for ensuring the reliability of distributed systems, including fault tolerance mechanisms, failure detection, and data exchange organization.

Research results: an analysis of failure causes in distributed systems was conducted, methods of failure detection and handling were studied, and modern approaches to building reliable systems, including microservices and serverless architectures, were reviewed.

The first section describes the theoretical foundations of reliability in distributed systems, key concepts, components, and principles of their construction, as well as basic communication services.

The second section analyzes the causes of system failures, methods of their detection and diagnosis, and data exchange technologies affecting system reliability.

The third section covers practical aspects of implementing reliable distributed systems, including platform technologies, multi-tier architectures, and evaluation of modern approaches such as serverless.

Conclusion: the application of modern reliability methods improves fault tolerance of distributed systems, ensures system continuity, and enhances data processing efficiency in complex information environments.

KEY WORDS: DISTRIBUTED SYSTEMS, RELIABILITY, FAULT TOLERANCE, MICROSERVICES, SERVERLESS, FAILURE DETECTION, DATA EXCHANGE, CLOUD COMPUTING.

ЗМІСТ

ВСТУП

У сучасному світі інформаційні технології відіграють ключову роль у функціонуванні більшості сфер діяльності, включаючи фінанси, телекомунікації, транспорт, медицину та електронну комерцію. Зростання обсягів даних, потреба у високій доступності сервісів та необхідність обробки запитів у реальному часі зумовлюють широке використання розподілених програмних систем. Такі системи дозволяють ефективно використовувати обчислювальні ресурси, забезпечують масштабованість та гнучкість, проте водночас створюють нові виклики, пов’язані з їх надійністю та відмовостійкістю.

Актуальність теми зумовлена необхідністю забезпечення стабільної та безперервної роботи розподілених систем в умовах високих навантажень, мережевих затримок, часткових відмов компонентів та інших факторів, що можуть негативно впливати на їх функціонування. Існуючі підходи до побудови розподілених систем не завжди повною мірою враховують складність взаємодії між компонентами, що може призводити до зниження продуктивності, втрати даних або повного припинення роботи сервісів. У зв’язку з цим виникає потреба у дослідженні та впровадженні ефективних методів забезпечення надійності, які дозволяють мінімізувати ризики відмов і підвищити стійкість систем до збоїв.

Метою роботи є дослідження та аналіз методів забезпечення надійності розподілених програмних систем, а також оцінка сучасних підходів до побудови відмовостійких і масштабованих програмних рішень.

Завданнями дослідження є аналіз предметної області розподілених систем, визначення основних понять і компонентів, дослідження принципів побудови надійних систем, аналіз причин виникнення збоїв і відмов, вивчення методів їх виявлення, діагностики та обробки, дослідження технологій обміну даними, а також аналіз сучасних архітектурних підходів, зокрема мікросервісної та serverless-архітектури. Крім того, передбачається розгляд платформних технологій та оцінка ефективності застосування різних підходів до забезпечення надійності.

Об’єктом дослідження є розподілені програмні системи як складні багатокомпонентні структури, що функціонують у мережевому середовищі.

Предметом дослідження є методи, моделі та інструменти забезпечення надійності розподілених систем, включаючи механізми відмовостійкості, резервування, реплікації, виявлення збоїв та організації взаємодії між компонентами системи.

Методи дослідження включають аналіз наукових і технічних джерел, порівняльний аналіз існуючих підходів, моделювання архітектурних рішень, а також узагальнення отриманих результатів для формування рекомендацій щодо підвищення надійності систем.

У першому розділі роботи розглянуто теоретичні основи забезпечення надійності розподілених систем, визначено ключові поняття, компоненти та принципи їх побудови, а також досліджено базові комунікаційні служби. У другому розділі проаналізовано причини відмов, методи їх виявлення та діагностики, а також технології обміну даними, що впливають на надійність систем. У третьому розділі розглянуто практичні аспекти реалізації надійних розподілених систем, сучасні платформні рішення, багаторівневі архітектури та проведено оцінку ефективності застосування сучасних підходів, зокрема serverless.

Очікується, що результати роботи сприятимуть підвищенню надійності розподілених програмних систем, зменшенню кількості відмов та забезпеченню стабільної роботи сервісів у різних умовах експлуатації.

Бакалаврська робота містить 90 сторінок, 12 рисунки, 1 таблицю, 3 розділи, список використаних джерел із 40 найменувань.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ЗАБЕЗПЕЧЕННЯ НАДІЙНОСТІ РОЗПОДІЛЕНИХ ПРОГРАМНИХ СИСТЕМ

1.1 Основні поняття та компоненти надійних розподілених систем

Надійні розподілені обчислювальні системи збираються з базових будівельних блоків. Найпростіше кажучи, це лише процеси та повідомлення, і якби наш інтерес був суто теоретичним , можливо, було б розумно на цьому зупинитися. З іншого боку, якщо ми хочемо застосувати теоретичні результати в практичних системах , нам потрібно буде працювати з досить детальним розумінням того, як насправді працюють практичні системи. У певному сенсі це прикро, оскільки реальні системи часто містять механізми, які мають недоліки, що здаються простими для виправлення, або несумісні один з одним, але мають таку довгу історію (або настільки глибоко вбудовані в стандарти), що може не бути жодного способу покращити поведінку, про яку йде мова. Однак, якщо ми хочемо насправді побудувати надійні розподілені системи, нереалістично наполягати на тому, що ми робитимемо це лише в ідеалізованих середовищах, які підтримують певну форму теоретично мотивованої структури. Реальний світ сильно відданий стандартам, і завдання перетворення наших теоретичних знань на практичні інструменти, які можуть взаємодіяти з цими стандартами, ймовірно, є найважливішим викликом, з яким стикається інженер комп'ютерних систем [1].

Розподілену систему прийнято вважати такою, що працює над багаторівневим набором мережевих сервісів (див. таблицю 1.1). Слід одразу зазначити, що нижчі рівні цієї ієрархії мають набагато більше сенсу, ніж верхні, і коли люди говорять про сумісність з ISO або про шари ISO, вони майже завжди мають на увазі рівні нижче «сеансу», а не сеансовий рівень або ті, що вище за нього. На жаль, протягом десятиліть державні закупівельні установи не розуміли цього і часто наполягали на «сумісності» з ISO. На щастя, більшість таких установ нарешті відмовилися від цієї мети та визнали, що чиста сумісність з ISO

не має сенсу, оскільки верхні рівні ієрархії не мають великого сенсу.

Таблиця 1.1.

Рівні протоколу OSI

Рівень	Опис
Прикладний (Application)	Програма, що використовує комунікаційне з'єднання
Представлення (Presentation)	Програмне забезпечення для кодування даних застосунку у повідомлення та їх декодування при отриманні
Сеансовий (Session)	Логіка, пов'язана із забезпеченням наскрізних властивостей, таких як надійність
Транспортний (Transport)	Програмне забезпечення, що відповідає за розбиття великих повідомлень на менші пакети
Мережевий (Network)	Функції маршрутизації, зазвичай обмежені малими або фіксованого розміру пакетами
Канальний (Data Link)	Протокол, що використовується для передачі та прийому пакетів
Фізичний (Physical)	Протокол, що визначає спосіб представлення пакетів у фізичному середовищі передачі

Кожен рівень відповідає програмній абстракції або апаратній функції та може бути реалізований у самій прикладній програмі, у бібліотеці процедур, з якою пов'язана програма, в операційній системі або навіть в апаратному забезпеченні комунікаційного пристрою. Як приклад, ось багаторівнева модель протоколу взаємодії відкритих систем (OSI) Міжнародної організації зі стандартизації (ISO) [2]:

• *Застосування:* Це сама прикладна програма, аж до моментів, коли вона виконує комунікаційні операції.

• *Презентація:* це програмне забезпечення, пов'язане з розміщенням даних у повідомленнях у форматі, який може бути інтерпретований процесом(ами) призначення, куди(яким) буде надіслано повідомлення, та для вилучення даних з повідомлень у процесі призначення.

• *Сесія:* це програмне забезпечення, пов'язане з підтримкою з'єднань між парами або наборами процесів.Сесія може мати властивості надійності та може вимагати певної форми ініціалізації або налаштування, залежно від конкретних параметрів, з якими працює користувач. У моделі OSI програмне забезпечення

сесії реалізує будь-які властивості надійності, а нижчим рівням ієрархії дозволено бути ненадійними, наприклад, втрачати повідомлення.

• *Транспорт:* Транспортний рівень відповідає за розбиття великих повідомлень на менші пакети, які враховують обмеження розміру, встановлені апаратним забезпеченням мережевого зв'язку. На стороні вхідного зв'язку транспортний рівень повторно збирає ці пакети в повідомлення, відкидаючи пакети, ідентифіковані як дублікати, або повідомлення, деякі складові пакети яких були втрачені під час передачі.

• *Мережа:* це рівень програмного забезпечення, що відповідає за маршрутизацію та низькорівневе керування потоком у мережах, що складаються з кількох фізичних сегментів, з'єднаних між собою так званими мостами та шлюзами.

• *Канал передачі даних:* Канальний рівень зазвичай є частиною апаратного забезпечення, яке реалізує комунікаційний пристрій. Цей рівень відповідає за надсилання та отримання пакетів, розпізнавання пакетів, призначених для локальної машини, та їх копіювання, відкидання пошкоджених пакетів та інші аспекти зв'язку на рівні інтерфейсу.

• *Фізичний:* Фізичний рівень відповідає за представлення пакетів на дроті, наприклад, за апаратну технологію передачі окремих бітів та протокол для отримання доступу до дроту, якщо його спільно використовують кілька комп'ютерів.

Корисно розрізняти типи гарантій, що надаються різними рівнями: *наскрізні* гарантії у випадку сеансового, презентаційного та прикладного рівнів, а також гарантії *«точка-точка»* для рівнів нижче. Це розмежування важливе у складних мережах, де повідомлення може потребувати пройти багато каналів, щоб досягти місця призначення [3]. У таких умовах властивість «точка-точка» – це властивість, яка діє лише на кожен стрибок – наприклад, протокол каналу передачі даних стосується одного стрибка, який виконує повідомлення, але не його загального маршруту або гарантій, яких програма може очікувати від самого каналу зв'язку. Сеансовий, презентаційний та прикладний рівні, навпаки,

нав'язують більш складну логічну абстракцію базовій мережі з властивостями, що діють між кінцевими точками каналу зв'язку, які можуть фізично поширюватися на складну підструктуру [4]. У частині III цієї роботи ми обговоримо дедалі складніші наскрізні властивості, поки нарешті не розширимо ці властивості до повністю охоплюючої розподіленої комунікаційної абстракції, яка охоплює розподілену систему в цілому та забезпечує узгоджену поведінку та гарантії протягом усього процесу. І, так само, як нашарування OSI будує свої наскрізні абстракції поверх абстракцій типу "точка-точка", нам потрібно буде будувати ці більш складні абстракції поверх того, що зрештою є властивостями типу "точка-точка".

Як видно на рисунку 1.1, кожен рівень логічно складається з логіки передачі та відповідної логіки прийому. На практиці це часто точно відповідає реалізації архітектури - наприклад, більшість сеансових протоколів працюють, нав'язуючи дію абстрактного керування кількома сеансами через спільне (або мультиплексоване) з'єднання канального рівня. Пакети, згенеровані різними сеансовими протоколами вищого рівня, можна розглядати як такі, що об'єднуються в єдиний потік пакетів, які обробляються IP-канальним рівнем як єдиний клієнт для своїх послуг [5].

Не слід вважати, що реалізація багаторівневої архітектури протоколів передбачає якийсь окремий модуль для кожного рівня. Дійсно, одна з причин, чому існуючі системи відхиляються від багаторівневої архітектури ISO, полягає в тому, що суворий стек протоколів на основі ISO був би досить неефективно в контексті сучасної операційної системи, де важливе повторне використання коду, а такі механізми, як IP-тунелювання, можуть захотіти повторно використовувати стек ISO «під» тим, що концептуально є другим екземпляром стеку [6]. І навпаки, для максимізації продуктивності функціональність багаторівневої архітектури часто стискається в один програмний елемент, а в деяких випадках шари можуть бути повністю обійдені для типів повідомлень, де рівень не вживатиме жодних дій - наприклад, якщо повідомлення дуже мале, транспортному рівню OSI не потрібно буде фрагментувати його на кілька

пакетів, і можна уявити собі спеціалізовану реалізацію стеку OSI, яка опускає транспортний рівень. Дійсно, переваги та недоліки багаторівневої архітектури протоколів стали головною темою дискусій в останні роки.

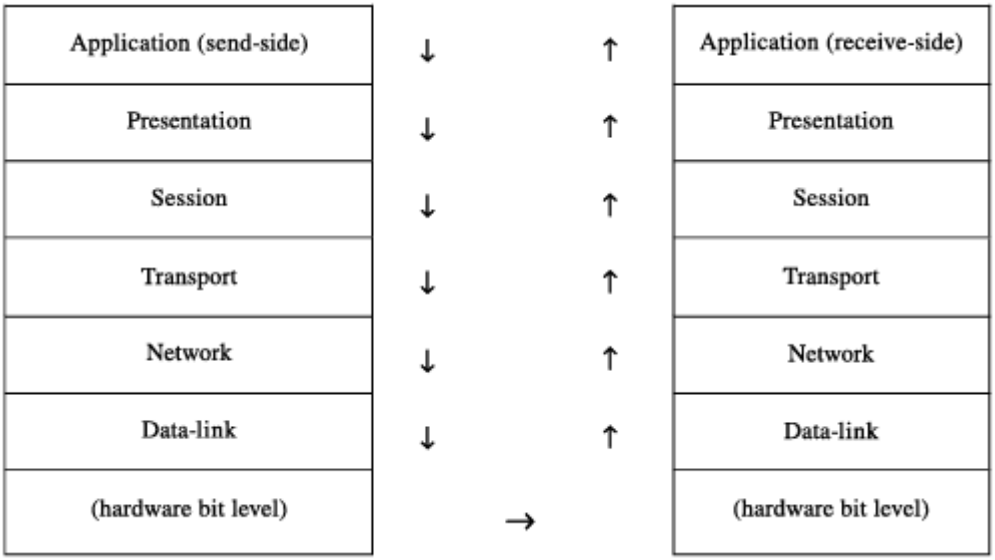

Рисунок 1.1 - Потік даних у стеку протоколів OSI

Хоча шарування OSI, ймовірно, є найвідомішою такою архітектурою, багатошарове комунікаційне програмне забезпечення є повсюдним, і існує багато інших прикладів багатошарових архітектур та багатошарових програмних систем. Далі в цій роботі ми розглянемо додаткові значення, в яких шарування OSI є застарілим, оскільки воно безпосередньо не стосується сеансів зв'язку з багатьма учасниками та не дуже добре поєднується з деякими новими типами комунікаційного обладнання, такими як системи комутації асинхронного режиму передачі (ATM) [7]. Обговорюючи цей момент, ми побачимо, що можна побудувати більш відповідні багатошарові архітектури, хоча вони не дуже точно відповідають шаруватості OSI. Таким чином, можна розглядати шарування як методологію, що відповідає конкретним рівням ієрархії OSI. Перша перспектива є популярною, яка набуває значення лише з впровадженням об'єктно-орієнтованих розподілених обчислювальних середовищ, які мають природну форму шарування, пов'язану з класами та підкласами об'єктів. Остання форма шарування, ймовірно, стала безнадійно несумісною зі стандартною практикою

на момент написання цієї роботи, хоча багато компаній та урядів продовжують вимагати, щоб продукти відповідали їй.

Можна стверджувати, що багаторівнева архітектура комунікації цінна, перш за все, як *описова абстракція* - модель, яка фіксує основну функціональність реальної системи зв'язку, але не обов'язково точно відображає її реалізацію. Ідея абстрагування поведінки розподіленої системи для її стислого опису або міркування про неї є дуже важливою. Однак, якщо абстракція не точно відповідає реалізації, це також створює низку проблем для розробника системи, який тепер зобов'язаний розробити специфікацію та доказ правильності абстракції; реалізувати, перевірити та протестувати відповідне програмне забезпечення; а також провести додатковий аналіз, який підтверджує, що абстракція точно моделює реалізацію [8].

Легко зрозуміти, як цей процес може зламатися - наприклад, майже неминуче, що зміни до реалізації доведеться вносити ще довго після розгортання системи. Якщо процес розробки справді такий складний, ймовірно, що аналіз загальної коректності не буде повторюватися для кожної такої зміни. Таким чином, з точки зору користувача, абстракції можуть бути палицею з двома кінцями. Вони пропонують привабливі та часто спрощені способи роботи зі складною системою, але вони також можуть бути спрощеними або навіть неправильними [9]. І це сильно впливає на загальну тему надійності. Певною мірою сам процес очищення компонента системи для його стислого опису може поставити під загрозу надійність складнішої системи, в якій цей компонент використовується.

Протягом усієї цієї роботи часто вдаватисямемо до моделей та абстракцій у набагато складніших ситуаціях, ніж нашарування OSI. Це допоможе нам міркувати про протоколи та порівнювати їх, а також доводити властивості складних розподілених систем. Водночас, однак, нам потрібно пам'ятати, що весь цей підхід вимагає своєрідного метапідходу, а саме вищого рівня абстракції, на якому ми можемо поставити під сумнів саму методологію, запитуючи, чи є методи, за допомогою яких ми створюємо надійні системи, самі можливим

джерелом ненадійності. Коли це виявиться так, нам потрібно зробити наступний крок, запитуючи, які систематичні засоби можна використовувати для боротьби з цими типами проблем надійності [10].

Чи можна побудувати добре структуровані розподілені обчислювальні системи, які можуть витримувати відмови власних компонентів або гарантувати інші види властивостей безпеки? У багаторівневих системах, таких як OSI, це питання насправді не вирішується, що є однією з причин, чому багаторівнева система OSI не працюватиме для наших цілей. Однак це питання є одним з найважливіших, які необхідно вирішити, якщо ми хочемо стверджувати, що ми розробили робочу методологію для проектування надійних розподілених обчислювальних систем [11]. Отже, методологія повинна вирішувати описові та структурні питання, а також практичні, такі як протоколи, що використовуються для подолання певного типу відмови або для координації певного типу взаємодії.

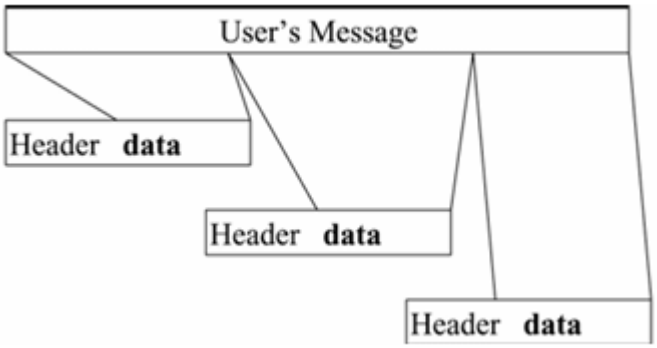

Рисунок 1.2 - Великі повідомлення фрагментуються для передачі

Найбазовішою комунікаційною технологією в будь-якій розподіленій системі є апаратна підтримка передачі повідомлень. Хоча існують деякі типи мереж, які пропонують спеціальні властивості, більшість сучасних мереж призначені для передачі даних у *пакетах* з певним фіксованим, але невеликим максимальним розміром. Кожен пакет складається із *заголовка,* який є структурою даних, що містить інформацію про пакет - його пункт призначення, маршрут тощо. Він містить *тіло,* яке є байтами, що складають вміст пакета. Також він може містити *трейлер,* який є другою структурою даних, що фізично

передається після заголовка та тіла і зазвичай складається з контрольної суми пакета, яку апаратне забезпечення обчислює та додає до нього як частину процесу передачі пакета.

Коли повідомлення користувача передається мережею, пакети, що фактично надсилаються по мережі, містять заголовки та трейлери, і можуть мати фіксований максимальний розмір.

Великі повідомлення надсилаються як кілька пакетів. Наприклад, на рисунку 1.2 показано повідомлення, фрагментоване на три пакети, кожен з яких містить заголовок та частину даних з оригінального повідомлення. Не всі схеми фрагментації включають трейлери, і на рисунку трейлер не показано.

Сучасне комунікаційне обладнання часто дозволяє великій кількості комп'ютерів використовувати одну комунікаційну структуру. З цієї причини необхідно вказати адресу, на яку слід передати повідомлення. Тому обладнання, яке використовується для зв'язку, зазвичай підтримує певну форму адресації , за допомогою якої можна ідентифікувати пункт призначення повідомлення [12]. Однак для більшості розробників програмного забезпечення важливішими є адреси, що підтримуються транспортними службами, доступними в більшості операційних систем. Ці *логічні адреси* є представленням розташування в мережі та використовуються для маршрутизації пакетів до пунктів призначення. Щоразу, коли пакет здійснює «стрибок» через канал зв'язку, комп'ютер-відправник повинен скопіювати апаратну адресу наступної машини на шляху у вихідний пакет. У цій роботі ми припускаємо, що кожен комп'ютер має логічну адресу, але мало що скажемо про апаратні адреси.

Читачі, знайомі з сучасними мережевими інструментами, знатимуть, що адреса, призначена комп'ютеру, може змінюватися з часом (особливо коли для динамічного призначення використовується протокол DHCP), що адреси можуть бути не унікальними (дійсно, оскільки сучасні брандмауери та транслятори мережевих адрес часто «зіставляють» внутрішні адреси, що використовуються в локальній мережі, із зовнішніми, видимими зовні, у співвідношенні багато до одного, повторне використання адрес є поширеним явищем), і що існують навіть

кілька стандартів адрес (IPv4 є найпоширенішим, а IPv6 - IPv6) [13].

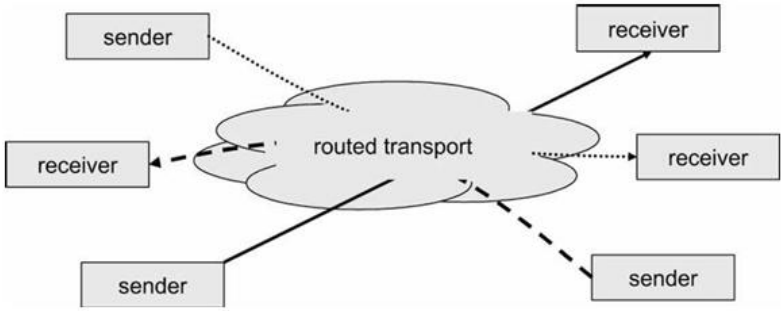

Рисунок 1.3 - Функціональність маршрутизації сучасного транспортного протоколу приховує топологію мережі від розробника застосунків

Для наших цілей у цій роботі ми відкладемо всі ці питання осторонь, а протоколи маршрутизації та проектування високошвидкісних мереж з накладанням залишимо як теми для іншого розгляду.

З іншого боку, існують дві функції адресації, які мають важливі наслідки для програмного забезпечення для зв'язку вищого рівня [14]. Це здатність програмного забезпечення (і часто базового мережевого обладнання) транслювати *та* розсилати *повідомлення* . Трансляція – це спосіб надсилання повідомлення таким чином, щоб воно було доставлено всім комп'ютерам, до яких воно доходить. Це можуть бути не всі комп'ютери в мережі через різні фактори, які можуть спричинити помилку отримання, але для багатьох цілей абсолютна надійність не потрібна. Щоб надіслати апаратне широкомовлення, прикладна програма зазвичай розміщує спеціальну логічну адресу у вихідному повідомленні, яку операційна система зіставляє з відповідною апаратною адресою. Повідомлення досягне лише тих машин, підключених до апаратного комунікаційного пристрою, на якому відбувається передача, тому використання цієї функції вимагає певних знань топології мережевого зв'язку [15].

Мультикаст – це форма широкомовлення, яка зв’язується з підмножиною комп’ютерів, підключених до комунікаційної мережі. Щоб використовувати мультикаст, зазвичай спочатку створюють нову групову адресу мультикасту та встановлюють її в апаратні інтерфейси, пов’язані з комунікаційним пристроєм.

Потім повідомлення мультикасту надсилаються так само, як і широкомовлення, але на апаратному рівні приймаються лише на тих інтерфейсах, яким було наказано встановити групову адресу, на яку призначене повідомлення. Багато мережевих маршрутизаторів та протоколів відстежують пакети мультикасту та пересилають їх автоматично, але це рідко робиться для пакетів широкомовлення.

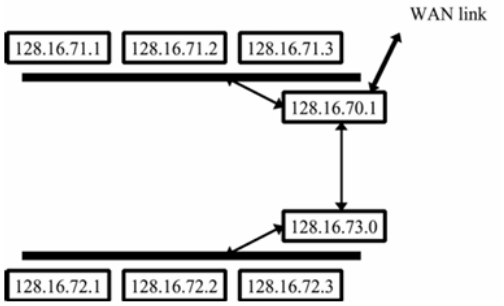

Рисунок 1.4 - Типова мережа може мати кілька взаємопов'язаних підмереж та «широкосмугове» з'єднання з Інтернетом.

Найцікавіші питання, які ми розглянемо в цій роботі, стосуються - середовищ програмування та інструментів, що знаходяться посередині, між прикладною програмою та комунікаційною інфраструктурою, для базової передачі повідомлень та підтримки надійних каналів.

Прикладами важливих служб проміжного програмного забезпечення є служба іменування, служби виявлення ресурсів, файлова система, служба часу та служби ключів безпеки, що використовуються для автентифікації в розподілених системах. Ми розглянемо всі ці служби детальніше пізніше, але коротко розглянемо їх тут для ясності [16].

Служба іменування – це набір доступних користувачеві каталогів, які відображають імена програм (або інші критерії вибору) на мережеві адреси комп’ютерів або програм. Служби іменування можуть відігравати багато ролей у розподіленій системі та є предметом інтенсивного дослідницького інтересу та швидкої еволюції. Коли ми обговорюємо іменування, ми побачимо, що все питання про те, що представляє ім’я, саме по собі є предметом значних дискусій та порушує важливі питання щодо концепцій абстракції та послуг у розподілених

обчислювальних середовищах. Надійність служби іменування включає такі питання, як довіра – чи можна довіряти службі іменування у правдивому відображенні імені на правильну мережеву адресу? Як можна знати, що об’єкт у кінці адреси – це той самий, про який говорила служба іменування? Це захопливі питання, і ми детально обговоримо їх пізніше в роботі.

Пов'язана тема стосується виявлення ресурсів [17]. У великих мережах спостерігається дедалі більший інтерес до підтримки механізмів самоконфігурації та самовідновлення. Наприклад, хотілося б, щоб універсальний контролер (для відеомагнітофонів, телевізорів тощо) міг автоматично виявляти медіа-пристрої в кімнаті, або щоб комп'ютер міг автоматично виявляти принтери поблизу. Деякі середовища програмування, такі як середовище JINI для програмістів Java, забезпечують певну функціональність ICQ («Я шукаю тебе»), хоча вони не є стандартними в інших типах інтернет-середовищ. Оскільки ми переходимо до світу з дедалі більшою кількістю комп'ютерів, новими видами невеликих мобільних пристроїв та вбудованим у середовище інтелектом, цей тип виявлення ресурсів стане важливою проблемою, і цілком ймовірно, що стандарти швидко з'являться. Зверніть увагу, що виявлення відрізняється від найменування: виявлення — це проблема пошуку ресурсів, що відповідають певним критеріям у цій області, а отже, створення списку імен. Іменування, з іншого боку, стосується правил призначення імен пристроям та зіставлення імен пристроїв з адресами.

Однак, з самого початку читач може врахувати, що якщо зловмисник проникне в систему та зможе маніпулювати зіставленням імен з мережевими адресами, то зможе вставити всілякі компоненти програмного забезпечення для стеження на шлях зв'язку від програми до служб, які вона використовує через мережу. Такі атаки зараз поширені в Інтернеті та відображають фундаментальну проблему, яка полягає в тому, що більшість технологій надійності мережі схильні довіряти механізмам найнижчого рівня, які зіставляють імена з адресами та направляють повідомлення до правильного хоста, коли їм надається адреса призначення.

Служба часу – це механізм для підтримки синхронізації годинників на наборі комп'ютерів та їх близькості до реального часу. Служби часу працюють над подоланням неточності недорогих годинників, що використовуються на багатьох типах комп'ютерів, і вони важливі в програмах, які або координують дії за допомогою реального часу, або використовують час для інших цілей, таких як обмеження терміну служби криптографічного ключа або додавання позначок часу до файлів під час їх оновлення. Багато можна сказати про час у розподіленій системі, і ми присвятимо значну частину цієї роботи питанням, що обертаються навколо всієї концепції "до" і "після" та її зв'язку з інтуїтивними концепціями часу в реальному світі. Очевидно, що надійність служби часу матиме важливі наслідки для надійності програм, які використовують час, тому служби часу та пов'язані з ними властивості надійності виявляться важливими в багатьох частинах цієї роботи [18].

Служби автентифікації, як не дивно, є новою технологією, якої бракує в більшості розподілених обчислювальних середовищ. Ці служби забезпечують надійні механізми для визначення того, хто надіслав повідомлення, для забезпечення того, щоб повідомлення міг прочитати лише призначений адресат, та для обмеження доступу до приватних даних, щоб можливий лише авторизований доступ [19]. Більшість сучасних обчислювальних систем розвинулися з періоду, коли контроль доступу був неформальним і базувався на основному принципі довіри між користувачами. Одним з дійсно серйозних наслідків є те, що розподілені системи, які хочуть накласти архітектуру безпеки або захисту на гетерогенне середовище, повинні подолати поширену тенденцію приймати запити, не ставлячи їх під сумнів, вірити інформації про ідентифікатор користувача, що міститься в повідомленнях, не перевіряючи її, та направляти повідомлення туди, куди вони забажають.

Якби банки працювали таким чином, можна було б підійти до касира в банку та передати йому аркуш паперу із запитом на список осіб, які мають рахунки у відділенні. Вивчивши відповідь та дізнавшись, що В. Гейтс є в списку, можна було б заповнити запит на баланс рахунку на ім'я В. Гейтса, запитуючи,

скільки грошей є на цьому рахунку. І після цього можна було б зняти частину цих грошей, в межах лімітів політики банку. Ні на якому етапі це не було б оскаржено: ідентифікація на різних аркушах паперу вважалася б довірливою для кожної операції. Така модель світу може здатися дивно довірливою, але саме з неї виникли сучасні розподілені обчислювальні системи.

1.2 Принципи та підходи до розробки надійних програмних рішень

Важлива тема, навколо якої зосереджена значна частина цієї роботи, стосується розробки універсальних інструментів, на основі яких можна створювати спеціалізовані розподілені системи. Такі інструменти можуть мати різні форми та бути суто концептуальними, наприклад, методологія чи теорія, яка пропонує корисне розуміння найкращого способу вирішення проблеми або може допомогти розробнику підтвердити, що запропоноване рішення матиме бажану властивість. Інструмент може пропонувати практичну допомогу на дуже низькому рівні, наприклад, усуваючи відносно механічні кроки, необхідні для кодування аргументів для виклику віддаленої процедури в повідомлення до сервера, який виконає дію [20]. Інструмент може втілювати складну поведінку вищого рівня, таку як протокол для виконання певної дії або подолання певного класу помилок. Інструменти можуть навіть виходити за рамки цього, роблячи наступний крок, пропонуючи механізми для контролю та управління програмним забезпеченням, створеним за допомогою інших інструментів.

Стало популярним говорити про розподілені системи, що підтримують *розподілені операційні середовища* — добре інтегровані набори інструментів, які можна використовувати разом один з одним для виконання потенційно складних завдань розподіленого програмування. Прикладами сучасного покоління середовищ розподіленого програмування є технологія .NET від Microsoft, Java Enterprise Edition (J2EE), система JINI від Sun для мобільних обчислень та CORBA. До старіших середовищ, які досі широко використовуються, належать середовище Open Network Computing (ONC) від Sun Microsystems, середовище

розподілених обчислень (DCE) від Open Software Foundation, і це лише неповний список. Деякі середовища особливо популярні для користувачів певних мов програмування — наприклад, спільнота Java схильна надавати перевагу J2EE, а спільнота C# (мова, майже ідентична Java) більше знайома з .NET. Програмісти на C++ схильні працювати з інструментами програмування, сумісними з CORBA. Поверх цих середовищ іноді можна знайти інструменти проміжного програмного забезпечення, які розширюють базове середовище додатковими функціями. Приклади, які будуть розглянуті в цьому тексті, включають Isis та Spread Toolkits — перший був розроблений мною та моїми колегами і буде розглянуто в розділі 21, тоді як другий — це більш сучасна система, розроблена в Університеті Джона Гопкінса Яїром Аміром, але з подібними функціями. (Це зовсім не повний список!)

Архітектури розподілених систем прагнуть вийти навіть за рамки концепції розподіленого обчислювального середовища. Архітектура — це загальний набір принципів проектування та стандартів реалізації, за якими можна розробити набір сумісних систем [21]. В принципі, кілька систем, що реалізують одну й ту саму архітектуру, будуть взаємодіяти, тому, якщо постачальники впроваджують конкуруючі рішення, отримане програмне забезпечення все ще можна об'єднати в єдину систему з компонентами, які навіть можуть взаємодіяти та співпрацювати один з одним. Незважаючи на появу .NET та J2EE, які на момент написання цієї роботи мають більш комерційне значення, Common Request Broker, або CORBA, ймовірно, все ще є найвідомішою архітектурою розподілених обчислень. CORBA корисна для побудови систем з використанням об'єктно-орієнтованого підходу, в якому системи розробляються як модулі, що взаємодіють. Таким чином, CORBA — це архітектура, а різні продукти на основі CORBA, які відповідають цій архітектурі, є розподіленими обчислювальними середовищами. .NET та J2EE — молодші брати CORBA; обидва успадковують безліч функцій безпосередньо від CORBA, а також підтримують дуже потужні механізми для створення програм, що мають доступ до баз даних — спеціалізація, якої бракувало CORBA донедавна, коли ця

архітектура почала поступатися цим новим розробникам [22].

Заглядаючи в майбутнє, багато аналітиків зараз стверджують, що найважливішою архітектурою з усіх будуть нові стандарти веб-сервісів, спрямовані на сприяння прямій взаємодії між комп'ютерами шляхом стандартизації всіх аспектів іменування, виклику, розуміння даних тощо. Хоча до цих прогнозів (як і до прогнозів «винтажу століття») слід ставитися зі скептицизмом, немає сумнівів, що веб-сервіси є надзвичайно амбітними та надзвичайно сумісними — розробники обіцяють, що ці стандарти вперше дозволять майже будь-чому спілкуватися майже з будь-чим іншим. Це лише перший крок: якщо ви розмовляєте французькою, а я англійською, я можу розмовляти з вами, але ви не обов'язково мене зрозумієте. Аналогічно, веб-сервіси повинні підтримуватися стандартами, скажімо, для спілкування з системою інвентаризації аптеки або запиту цінової пропозиції на партію деталей машин. Тим не менш, такі стандарти, безумовно, можна визначити в багатьох умовах. Постачальники, які підтримують .NET та J2EE, рекламують легкість створення систем веб-сервісів під час використання своїх продуктів, і дуже багато постачальників оголосили про плани підтримки цієї архітектури. З іншого боку, ця архітектура погано підходить для деяких цілей: веб-сервіси мають нечітке уявлення про надійність і точно не підійдуть для побудови систем з дуже високою доступністю (принаймні, протягом перших кількох років) або для підтримки таких програм, як дуже масштабні системи моніторингу інформації або масштабні сенсорні архітектури. Дійсно, можливо, найкраще розглядати веб-сервіси (наразі) як ймовірного переможця в битві за архітектури, за допомогою яких клієнт підключається до однієї бази даних за раз, хоча, можливо, як частину довшої серії операцій, що включають транзакції на кількох базах даних (наприклад, для купівлі авіаквитка, потім бронювання номера в готелі, потім бронювання автомобіля тощо). Для таких конвеєрних, відносно асинхронних програм веб-сервіси здаються видатною розробкою. Якби хтось зосередився на розробці нової військової платформи, спрямованої на інтеграцію всіх пристроїв, керованих комп'ютером, на полі бою, веб-сервіси здавалися б набагато менш

доречними для цієї потреби.

Можна було б очікувати, що кінцевою точкою для багаторівневої розподіленої системної архітектури буде рівень програми, але це не обов'язково так. Розподілена програма також може бути певним типом служби операційної системи, побудованої на основі комунікаційних інструментів, які ми обговорювали, наприклад, розподілена файлова система є додатком у сенсі багаторівневого розподілу OSI, але користувач обчислювальної системи може думати про файлову систему як про службу операційної системи, поверх якої можна визначати та виконувати програми. Отже, в рамках багаторівневого розподілу OSI програма - це будь-яке окреме рішення чітко визначеної проблеми, яке представляє для своїх користувачів щось інше, ніж абстракція прямої комунікації. Розподілена файлова система - це лише один із багатьох прикладів. Інші включають технології шин повідомлень, розподілені системи баз даних, електронну пошту, мережеві дошки оголошень та Інтернет [23]. У найближчому майбутньому, ймовірно, з'являться комп'ютерні системи спільної роботи, сенсорні мережі та мультимедійні цифрові бібліотечні системи як подальші приклади в цій галузі.

Навмисне обмеження такої багаторівневої структури, як ієрархія OSI, полягає в тому, що вона насправді не розрізняє такі типи програм, які надають послуги розподіленим програмам вищого рівня, від того, що можна назвати рішеннями для кінцевого користувача, а саме програм, які працюють на рівні зв'язку для безпосередньої реалізації команд для людини. Хотілося б вірити, що розподілена система управління повітряним рухом має набагато більше структури, ніж програма передачі файлів, проте ієрархія OSI розглядає обидві системи як приклади програм. Нам бракує хорошої системи класифікації для різних типів розподілених програм [24].

Фактично, навіть складні розподілені програми можуть бути лише компонентами ще більш масштабних розподілених систем — легко уявити собі розподілену систему, яка використовує інструментарій розподілених обчислень для інтеграції програми, що використовує розподілені файли, з програмою, що

зберігає інформацію в розподіленій базі даних. У середовищі управління повітряним рухом доступність може бути настільки критичною, що доведеться одночасно запускати кілька копій програмного забезпечення, причому одна версія резервує іншу. Тут вся система управління повітряним рухом на одному рівні є складною розподіленою програмою сама по собі, але на іншому метарівні є лише компонентом загальної структури надійності, видимої в масштабі сотень комп'ютерів, розташованих у кількох центрах управління повітряним рухом [25].

Ці спостереження вказують на необхідність подальших досліджень архітектур для розподілених обчислень, особливо в галузях, що стосуються високої надійності. Завдяки кращим архітектурним інструментам ми могли б ефективніше міркувати про великі, складні системи, такі як ті, що щойно згадані. Більше того, архітектурні стандарти заохочували б постачальників пропонувати інструменти для підтримки моделі. Однак сьогодні зрілість цієї галузі досягла рівня, на якому як усвідомлення проблеми ширше, так і варіанти, доступні для її вирішення, краще зрозумілі. Було б дуже шкода, якби спільнота, що розробляє архітектури веб-сервісів, втратила цей унікальний шанс по-справжньому вирішити цю нагальну потребу. Під час підготовки цієї редакції було багато причин для натхнення, зокрема, ціла низка «груп спеціальних інтересів» веб-сервісів, зосереджених на цих галузях. Однак, обговорення – це одне, а широко прийнятий стандарт – зовсім інше. Тим з нас, хто шукає зрілі стандарти, підкріплені різноманітними конкуруючими інструментами та продуктами, потрібно буде спостерігати, чекати та сподіватися, що ці зусилля будуть успішними.

1.3 Базові комунікаційні служби та протоколи в розподілених системах

Стандарт зв'язку - це набір специфікацій, що регулюють типи повідомлень, які можна надсилати в системі, формати заголовків і трейлерів повідомлень, правила кодування для розміщення даних у повідомленнях, а також

правила, що регулюють формат і використання адрес джерела та призначення. Крім того, стандарт зазвичай визначає низку протоколів, які повинен реалізувати постачальник.

Прикладами комунікаційних стандартів, які широко використовуються, хоча й не повсюдно, є:

Інтернет-протоколи: Ці протоколи виникли в результаті роботи, виконаної Агентством передових дослідницьких проектів Міністерства оборони США (DARPA) у 1970-х роках, і поступово переросли в широкомасштабну високопродуктивну мережу, що з'єднує мільйони комп'ютерів. Протоколи, що використовуються в Інтернеті, включають IP, базовий пакетний протокол, а також UDP, TCP та IP-мультиадресацію, кожен з яких є протоколом вищого рівня, що накладається на IP. З появою Інтернету Інтернет вибухово розвивався з середини 1990-х років.

• *SOAP (Простий протокол доступу до об'єктів):* Цей протокол було запропоновано як частину набору стандартів, пов'язаних з веб-сервісами, архітектурною основою, за допомогою якої комп'ютери взаємодіють з іншими комп'ютерами так само, як браузер взаємодіє з веб-сайтом. Повідомлення SOAP кодуються в XML, повністю текстовому форматі, і тому є дуже багатослівними та досить великими порівняно з іншими представленнями. З іншого боку, стандарт підтримується величезною кількістю постачальників.

• *Власні стандарти:* Багато постачальників використовують власні стандарти у своїх продуктах, і зазвичай їх документують, щоб клієнти та інші постачальники могли створювати сумісні програми.

Але те, що щось є стандартом, не означає, що воно буде широко прийняте. Наприклад, ось деякі широко цитовані стандарти, які ніколи не обмежують комерційну гірчицю:

• *Протоколи взаємодії відкритих систем:* ці протоколи схожі на набір інтернет-протоколів, але використовують стандарти та конвенції, що виникли в організації ISO. Ми згадуємо їх лише тому, що Європейський Союз зобов'язав використовувати ці протоколи у 1980-х роках. Насправді вони не мають

належної підтримки, і розробники, які вирішили їх використовувати, опиняються у скрутному становищі, коли їм доводиться дотримуватися стандарту, але водночас відмовлятися від домінуючого технологічного набору для всього Інтернету [26]. На практиці ЄС за останні два десятиліття надав так багато винятків зі своїх вимог, що їх фактично ігнорують - повчальна історія для тих, хто вважає, що просто тому, що щось є «стандартним», воно краще за альтернативи, які мають більшу комерційну підтримку.

• *Стандарт безпеки Помаранчевої книги:* Визначена Міністерством оборони США, Помаранчева книга визначала правила управління захищеною інформацією у військових та урядових системах США. Стандарт був дуже детальним і фактично був обов'язковим у Сполучених Штатах протягом багатьох років. Проте майже кожна система, фактично придбана військовими, була звільнена від дотримання вимог, оскільки ширша галузь просто не визнавала необхідності такого виду безпеки та ставила під сумнів неявні припущення, що лежали в основі фактичної технічної пропозиції. На момент написання цієї роботи Помаранчева книга, здається, померла.

Протягом 1990-х років відкриті системи, а саме системи, в яких комп'ютери різних постачальників могли запускати незалежно розроблене програмне забезпечення, стали важливою тенденцією, завойовуючи більшість ринку серверних систем; одночасно власна операційна система Windows від Microsoft стала домінувати на настільних комп'ютерах. Ці дві паралельні тенденції поставили перед розробниками та дизайнерами протоколів дилему. Сьогодні ми бачимо поєднання широко прийнятих стандартів в Інтернеті в цілому та власних стандартів (таких як протокол доступу до файлової системи Microsoft), що використовуються в локальних мережах. У багатьох відношеннях можна припустити, що домінували рішення "найкращого класу". Наприклад, протоколи віддаленого доступу до файлів Microsoft загалом вважаються перевершуючими протоколи NFS, з якими вони спочатку конкурували. Однак існує також значний елемент випадковості. Справа в тому, що будь-яка технологія розвивається з часом. Windows та відкриті (Linux) спільноти по суті

слідували одна за одною, кожна з яких імітувала будь-які успішні інновації, запроваджені іншою. Фактично, ринкові сили виявилися набагато важливішими, ніж сам факт стандартизації або закони, що застосовуються на одному закритому ринку чи іншому.

Основною рушійною силою на користь стандартів була *сумісність* . Тобто, користувачам обчислювальних машин потрібні способи взаємодії основних програм та створення нових програм, які мають доступ до старих. Протягом тривалого часу сумісність була переважно проблемою, що спостерігалася в спільноті баз даних, але тепер ця тенденція поширилася також на ширшу сферу мереж та розподілених систем. Протягом наступного десятиліття здається ймовірним, що найважливіші інновації частково виникнуть завдяки раптовим покращенням сумісності, ідеї, яку можна простежити до спільноти CORBA [27]. Спочатку спільнота Java приділяла сумісності дещо меншу увагу через її ранню зосередженість на «єдиній універсальній мові». В останні роки сумісність відстоювала Microsoft у своїй лінійці продуктів .NET, і цей акцент незабаром мотивував спільноту J2EE також докласти більше зусиль до цього питання. Сьогодні однією з найважливіших причин існування стандартів є сприяння сумісності.

Архітектура веб-сервісів прагне досягти цієї мети, принаймні для основних питань зіставлення клієнтської програми з необхідними їй сервісами та надання їй можливості викликати операції незалежно від платформи та розташування. Для цієї останньої мети використовується протокол SOAP, що надає SOAP унікально центральну роль у майбутньому поколінні розподілених програм. SOAP – це багаторівневий стандарт: він виражає правила для кодування запиту, але не для передачі повідомлень від клієнта до сервера. У найпоширенішому багаторівневому використанні SOAP працює поверх HTTP, стандартного протоколу для зв'язку з веб-сайтом, а HTTP, у свою чергу, працює поверх TCP, який генерує IP-пакети.

У решті цього розділу коротко розглядається кожен із цих компонентів. У нашому розгляді не розглядаються всілякі інші потенційно релевантні

стандарти, і він призначений скоріше для «ознайомлення» з темою, ніж для будь-якого вичерпного розгляду. Читачеві пропонується звернутися до веб-сайтів, що підтримуються консорціумом W3 та такими постачальниками, як IBM або Microsoft, для отримання детальної інформації про те, як поводяться SOAP та HTTP. Детальну інформацію про те, як реалізовано пакет IP-протоколів.

У цьому розділі представлено три основні компоненти набору інтернет-протоколів: протокол IP, на якому базуються інші, та протоколи TCP і UDP, які зазвичай використовуються програмами. Ми також обговоримо деякі нещодавні розширення рівня протоколу IP для підтримки протоколів багатоадресної розсилки IP. Ведеться значне обговорення безпеки рівня IP, але на момент написання цієї роботи жодна пропозиція не отримала широкого визнання, і ми дуже мало розповімо про цю поточну роботу з міркувань стислості.

Найнижчий рівень набору інтернет-протоколів – це протокол передачі пакетів без встановлення з'єднання, який називається IP. IP відповідає за ненадійну передачу пакетів змінного розміру від машини відправника до машини призначення. Хоча IP насправді може підтримувати досить великі розміри пакетів, зазвичай існує верхня межа в 1518 байт. IP-пакети повинні відповідати фіксованому формату, що складається із заголовка пакета змінної довжини та тіла змінної довжини. Фактична довжина заголовка, тіла та кінцевого пакета визначається за допомогою полів довжини, які розташовані з фіксованими зміщеннями в заголовку. Очікується, що програма, яка безпосередньо використовує IP, форматуватиме свої пакети відповідно до цього стандарту. Однак пряме використання IP зазвичай обмежується операційною системою через проблеми безпеки, що виникають через можливість використання програмами такої функції для імітації певного стандартного протоколу, такого як протокол керування передачею (TCP), роблячи це нестандартним чином, що може порушити роботу віддалених машин або створити лазівки в безпеці [28].

Реалізації IP зазвичай забезпечують функціональність маршрутизації, використовуючи статичну або динамічну архітектуру маршрутизації. Тип маршрутизації, що використовується, залежатиме від складності інсталяції та

конфігурації програмного забезпечення Інтернету, і ця тема виходить за рамки цієї роботи.

У 1995 році IP було вдосконалено, щоб забезпечити архітектуру безпеки, за допомогою якої корисне навантаження пакетів може бути зашифроване, щоб запобігти визначенню вмісту пакетів зловмисниками, а також надати опції для підписів або інших даних автентифікації в трейлері пакета.

TCP – це назва протоколу, орієнтованого на з'єднання, в рамках набору інтернет-протоколів. Користувачі TCP починають зі встановлення TCP-з'єднання, яке здійснюється шляхом налаштування однієї програми на прослуховування та прийняття вхідних з'єднань, поки інша програма підключається до неї. TCP-з'єднання гарантує, що дані будуть доставлені в порядку надсилання, без втрат або дублювання, і повідомлятиме про кінець файлу, якщо процес на будь-якому кінці виходить з каналу або закриває його. TCP-з'єднання орієнтовані на байтовий потік: хоча програма, що відправляє, може надсилати блоки байтів, базова модель зв'язку розглядає цей зв'язок як безперервну послідовність байтів [29]. Таким чином, TCP дозволяється втрачати інформацію про межі між повідомленнями, так що те, що логічно є одним повідомленням, може бути доставлено кількома меншими порціями або доставлено разом з фрагментами попереднього або наступного повідомлення (проте завжди зберігаючи порядок байтів). Якщо передаються дуже малі повідомлення, TCP трохи затримує їх, щоб спробувати заповнити більші пакети для ефективної передачі; користувач повинен вимкнути цю поведінку, якщо потрібна негайна передача.

Програми, що передбачають одночасне використання TCP-з'єднання, повинні блокуватися, запобігаючи можливості одночасного виконання кількох операцій запису на одному каналі; якщо це станеться, то дані від різних записувачів можуть чергуватися, коли канал заповниться.

UDP – це протокол, орієнтований на повідомлення або дейтаграми. За допомогою цього протоколу програма надсилає повідомлення, які зберігаються у відправленому вигляді та доставляються до пункту призначення в

недоторканому вигляді або взагалі не доставляються. Не потрібне з'єднання, і немає жодних гарантій, що повідомлення дійде, що повідомлення будуть доставлені в певному порядку, або навіть що не виникнуть дублікати. Операційна система встановлює обмеження розміру для кожного сокета UDP, зазвичай 8 КБ. Таким чином, програма, якій потрібно надіслати велике повідомлення, повинна збільшити це обмеження, контрольоване ОС, або розбити повідомлення на кілька пакетів [30]. Крім того, існує обмеження максимально можливого розміру пакета UDP у 64 КБ.

Внутрішньо UDP зазвичай фрагментує повідомлення на менші частини, які точно відповідають максимальному розміру пакета, який Ethernet може передати в одному обладнанні.

Якщо розмір UDP-пакета перевищує максимальний, він надсилається як серія менших IP-пакетів. Після отримання вони знову збираються в більший пакет. Якщо будь-який фрагмент втрачається, UDP-пакет зрештою відкидається.

Читач може задатися питанням, чому використовується така дворівнева схема фрагментації - чому б просто не обмежити UDP до 1518 байт? Щоб зрозуміти цю схему, корисно почати з вимірювання вартості, пов'язаної з викликом комунікаційної системи. У типовій операційній системі така операція має мінімальні накладні витрати від двадцяти тисяч до п'ятдесяти тисяч інструкцій, незалежно від розміру об'єкта даних, що передається. Отже, ідея полягає в тому, щоб уникнути багаторазового проходження довгих шляхів коду в операційній системі. Коли передається UDP-пакет розміром 8 КБ, код для його фрагментації на менші частини виконується глибоко в операційній системі. Це може заощадити десятки тисяч інструкцій. З іншого боку, не докладається жодних зусиль для забезпечення проходження IP-пакетів, а UDP-пакети споживають дефіцитний ресурс: пам'ять ядра. Таким чином, не хочеться, щоб обмеження розміру UDP-пакета стало занадто великим, оскільки зростає ризик втрати принаймні одного IP-фрагмента (у цьому випадку весь UDP-пакет потрібно відкинути), а також тому, що обсяг необхідної пам'яті став би непомірним. Хоча 8 КБ здається дещо малим за сучасними стандартами, це стало

прийнятим обмеженням розміру за замовчуванням наприкінці 1980-х років.

Можна також задатися питанням, чому зв'язок взагалі має бути таким дорогим. Насправді, це дуже цікава та досить актуальна тема, особливо з огляду на нещодавню роботу, яка знизила вартість надсилання повідомлення (на деяких платформах) до всього шести інструкцій. У цьому підході, який називається *Активні повідомлення*, операційна система повністю виключається зі шляху передачі повідомлень, і якщо хтось готовий заплатити трохи вищу ціну, аналогічна перевага можлива навіть у більш стандартній архітектурі зв'язку. Комерційні операційні системи, що пропонують порівняно низьку затримку та високу пропускну здатність, стали доступними наприкінці 1990-х років. Однак середня операційна система точно не наздожене передові підходи протягом багатьох років [31]. Таким чином, додаткам, можливо, доведеться продовжувати жити з величезними та фактично непотрібними накладними витратами на даний момент.

IP-мультиадресація була відносно пізнім доповненням до набору інтернет-протоколів. За допомогою IP-мультиадресації повідомлення UDP або IP можна передавати групам адресатів, на відміну від одного пункту призначення "точка-точка". Такий підхід розширює можливості багатоадресної розсилки інтерфейсу Ethernet для роботи навіть у складних мережах з маршрутизацією та мостами між сегментами Ethernet.

IP-мультиадресація – це сеанс-орієнтований протокол: перед початком зв'язку потрібна певна робота. Процеси, які будуть взаємодіяти, повинні створити IP- адресу мультиадресації, яка є інтернет-адресою класу D, що містить ідентифікатор мультиадресації в молодших 28 бітах. Ці процеси також повинні узгодити єдиний номер порту, який усі використовуватимуть для сеансу зв'язку. Під час запуску кожен процес встановлює IP-адресу у свою локальну систему, використовуючи системні виклики, які розміщують IP-адресу мультиадресації на інтерфейсі(ах) Ethernet, до якого підключено машину. Таблиці маршрутизації, що використовуються IP, про які детальніше розповідається нижче, також оновлюються, щоб гарантувати, що IP-пакети мультиадресації будуть

пересилатися до кожного пункту призначення та мережі, в якій знаходяться члени групи [32].

Після завершення цього налаштування, IP-мультиадресація ініціюється простим надсиланням UDP-пакета з адресою групи IP-мультиадресації та номером порту. Коли цей пакет досягає машини, що входить до списку призначення, створюється його копія та доставляється локальним програмам, які отримують пакети на цьому порту. Якщо кілька пакетів прив'язані до одного порту на одній машині, для кожної з них створюється копія.

Як і UDP, IP-мультикаст є ненадійним протоколом: пакети можуть бути втрачені, дубльовані або доставлені не в порядку, і не всі члени групи побачать однакову схему втрати та доставки. Таким чином, хоча можна створити надійні протоколи зв'язку через IP-мультикаст, сам протокол за своєю суттю є ненадійним.

Під час використання через інтерфейс UDP, функція багатоадресної розсилки UDP подібна до функції дейтаграм UDP, оскільки кожен пакет може мати довжину, рівну максимальному розміру передач UDP, який зазвичай становить 8 КБ. Однак, під час надсилання багатоадресної розсилки IP або UDP важливо пам'ятати, що спостережувана надійність може відрізнятися від пункту призначення до пункту призначення. Одна машина може отримати пакет, який інші втрачають через обмеження пам'яті або пошкодження, спричинене слабким сигналом на середовищі зв'язку, і втрата навіть одного фрагмента великого повідомлення UDP призведе до втрати всього повідомлення. Будь-яка програма, яка використовує цей транспортний протокол, повинна ретельно контролювати коефіцієнти втрат, оскільки ефективна продуктивність для малих повідомлень може бути насправді кращою, ніж для великих через це обмеження.

Коли велика кількість відправників одночасно надсилає IP-мультикаст-пакети, іноді спостерігаються епізоди дуже високого рівня втрат. Вони називаються «широкомовними штормами» або «мультикаст-штормами», і виникають, коли пропускна здатність перевантажується сплесками вхідних пакетів, що надходять один за одним. Тому програми, що працюють на IP-

мультикасті, повинні обмежувати себе швидкістю передачі даних нижчою за поріг, за якого це може статися.

Архітектура, описана вище, дозволяє втрачати пакети на кожному стрибку в підсистемі зв'язку. Якщо пакет проходить багато стрибків, ймовірність втрати, ймовірно, зростатиме пропорційно, що призведе до лінійного зниження надійності мережі з діаметром мережі. Існує альтернативний підхід, за якого виправлення помилок виконуватиметься стрибком за стрибком. Хоча пакети все ще можуть бути втрачені, якщо проміжна машина вийде з ладу, такий підхід матиме значно знижені показники втрат за певних невеликих, але фіксованих фонових витрат (коли ми обговорюватимемо деталі надійних протоколів зв'язку, ми побачимо, що накладні витрати не обов'язково повинні бути дуже високими). Чому ж тоді більшість систем віддають перевагу підходу, який, ймовірно, є набагато менш надійним?

У цій роботі порівнювали протоколи надійності "від початку до кінця", які працюють лише між джерелом і одержувачем повідомлення, з протоколами надійності "перехід за перехідником". Вони стверджували, що навіть якщо надійність маршрутизованої мережі покращується завдяки використанню протоколів надійності "перехід за перехідником", вона все одно не буде достатньо високою, щоб повністю подолати втрату пакетів. Пакети все ще можуть бути пошкоджені шумом на лініях, машини можуть вийти з ладу, а динамічні зміни маршрутизації можуть перекидати пакет, доки він не буде відкинутий. Більше того, вони стверджували, що виміряні середні показники втрат для мереж з легким або помірним навантаженням надзвичайно низькі. Щоправда, маршрутизація наражає пакет на повторювані загрози, але загальна надійність маршрутизованої мережі все одно буде в середньому дуже високою, причому найгірший випадок буде переважати через такі події, як оновлення таблиці маршрутизації та збої, які виправлення помилок "перехід за перехідником" не подолає. З цього автори роблять висновок, що оскільки методи покрокової надійності збільшують складність і знижують продуктивність, і все ще повинні дублюватися наскрізними механізмами надійності, можна було б

використовувати простіший і швидший протокол зв'язку на рівні каналу. Це аргумент наскрізного зв'язку, який став одним із визначальних принципів, що регулюють сучасне проектування мереж.

Робота зосереджена на конкретному прикладі, що стосується протоколу передачі файлів (FTP). У роботі зазначається, що використаний аналіз багато в чому пов'язаний з прикладом та фактичними властивостями надійності відповідних ліній зв'язку. Більше того, Сальцера цікавила саме надійність механізму передачі пакетів: частота відмов та впорядкування. Ці моменти важливі, оскільки багато авторів почали цитувати аргумент "кінець-до-кінця" набагато розширеніше, стверджуючи, що це абсолютний аргумент проти розміщення будь-якої форми власності чи гарантії в підсистемі зв'язку. Пізніше ми обговоримо протоколи, які *потребують* розміщення властивостей та гарантій у підсистемах як спосіб забезпечення загальносистемних властивостей, які інакше не були б досяжні. Таким чином, ті, хто приймає узагальнений аргумент "кінець-до-кінця", схильні виступати проти використання таких протоколів з філософських (схильні сказати "релігійних") підстав.

Більш зрілий погляд полягає в тому, що наскрізний аргумент є однією з тих ситуацій, коли слід прийняти дуже загальне розуміння, але з певною часткою скептицизму. З одного боку, наскрізний аргумент є явно правильним у ситуаціях, коли можливий аналіз [33]. Однак наскрізний аргумент не можна застосовувати сліпо: є ситуації, коли низькорівневі властивості є корисними та дійсно знижують складність і вартість прикладного програмного забезпечення, і для цих ситуацій чистий наскрізний підхід може бути недоречним, що призводить до складніших застосунків, які схильні до помилок або, на практиці, неможливі для створення.

У мережі з високим рівнем втрат на рівні каналів або в мережі, яка має серйозний ризик нестачі пам'яті, якщо не використовується керування потоком між каналами, наскрізний підхід може призвести до майже повної втрати пакетів, тоді як схема, яка виправляє втрату пакетів і здійснює керування потоком на рівні каналу, може забезпечити прийнятну продуктивність. Отже, це випадок, коли

аналіз Сальцера можна застосувати так, як він його спочатку сформулював, але це призведе до іншого висновку. Коли ми розглянемо протоколи надійності, представлені в розділі III цієї роботи, ми побачимо, що певні форми послідовної розподіленої поведінки (такі, як це необхідно в відмовостійкій когерентній схемі кешування) залежать від загальносистемної угоди, яка має бути стандартизована та інтегрована з механізмами низькорівневого повідомлення про збої . Виключення такого механізму з рівня передачі просто змушує програміста додатка створювати його як частину додатка; якщо середовище програмування має бути загальним і розширюваним, це може означати, що механізм робиться частиною середовища або повністю відмовляється від нього. Таким чином, коли ми розглядаємо розподілені середовища програмування, такі як архітектура CORBA, що обговорювалася в розділі 5, насправді існує базовий вибір дизайну. Або така функція робиться частиною архітектури, або, якщо її опустити, жодна програма не може досягти такого типу узгодженості загальним та сумісним способом, окрім як стосовно інших програм, реалізованих тією ж командою розробників. Ці приклади ілюструють, що, як і багато інженерних аргументів, наскрізний підхід є дуже доречним у певних ситуаціях, але не завжди.

1.4 Висновок по розділу

У першому розділі було розглянуто теоретичні основи забезпечення надійності розподілених програмних систем. Досліджено основні поняття, архітектурні компоненти та характеристики надійних розподілених систем, а також принципи їх побудови. Проаналізовано підходи до розробки програмних рішень, орієнтованих на високу доступність, масштабованість та відмовостійкість. Окрему увагу приділено базовим комунікаційним службам і протоколам, які забезпечують взаємодію компонентів у розподіленому середовищі. Встановлено, що надійність системи значною мірою залежить від правильно обраної архітектури, способів обміну даними та механізмів координації між вузлами.

РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ ЗАБЕЗПЕЧЕННЯ НАДІЙНОСТІ РОЗПОДІЛЕНИХ СИСТЕМ

2.1 Причини відмов у розподілених комп'ютерних системах

У цій частині ми розглядатимемо технології для забезпечення надійності реальних розподілених систем. Перш ніж взятися за це завдання, буде корисно коротко зрозуміти причини, чому розподілені системи виходять з ладу. Хоча існують деякі вражаючі дослідження наслідків збоїв, наше розгляд спирається переважно на роботу, яка вивчала питання про *причини* збоїв систем, працюючи в Tandem Computers, а також на презентації, розробника транзакційної системної архітектури Tandem. Усі дослідники зосередилися на системах, розроблених для максимальної надійності, і могли б зробити інші висновки, якби вони розглядали великі розподілені системи, що включають технології, побудовані з менш суворими стандартами надійності. На жаль, схоже, що було проведено відносно мало формальних досліджень щодо частоти відмов та причин їх виникнення в системах, які *не були* спроектовані з урахуванням надійності як основної мети, незважаючи на те, що велика кількість систем, що використовуються в критичних умовах, містять компоненти з цією властивістю.

Апаратні збої були домінуючим фактором при розробці надійних систем до кінця 1980-х років. Апаратне забезпечення може виходити з ладу різними способами, але з удосконаленням електронної упаковки та збільшенням щільності інтегральних схем надійність апаратного забезпечення значно зросла. Це покращення надійності відображає зменшення тепловиділення та споживання енергії меншими схемами, зменшення кількості зовнішніх з'єднань та проводів, а також удосконалення технологій виробництва. Наслідком цього є те, що простої системи, пов'язані з апаратним забезпеченням, швидко стають незначним компонентом загальних проблем надійності, з якими стикаються у великій, складній розподіленій системі. Очевидно, що апаратний збій залишається фактором, особливо на невеликих портативних пристроях,

пристроях, що залежать від батареї, та ноутбуках або настільних комп'ютерах, до яких, як правило, ставляться більш грубо, ніж до серверів. Однак частота апаратних збоїв загалом знижується, особливо на серверних платформах.

Оскільки апаратні збої залишаються значною проблемою надійності сьогодні, спостережувані проблеми найчастіше пов'язані з внутрішніми обмеженнями роз'ємів та механічних пристроїв. Таким чином, проблеми з комп'ютерною мережею (що проявляються у втраті повідомлень або збоях розділення, коли компонент системи відключається від якогось іншого компонента) займають високе місце у списку причин збоїв, пов'язаних з обладнанням, для будь-якої сучасної системи. Збої дисків також є основною причиною простою в системах, залежних від великих файлових серверів або серверів баз даних, хоча дискові масиви типу RAID можуть значною мірою захистити від таких проблем. Звичайно, навіть RAID-диски виходять з ладу (досить часто через нерозумні помилки ремонту, такі як вилучення неправильного модуля під час обслуговування RAID-блоку, який зазнав єдиного збою) [34]. Однак, кількість збоїв дисків усіх видів зменшилася щонайменше на порядок порівняно з ситуацією, що спостерігалася в 1980-х роках.

Поширене джерело простоїв, пов'язане з апаратним забезпеченням, мало пов'язане з відмовами, хоча воно може серйозно вплинути на доступність системи та її сприйняття як надійність. Будь-яка критично важлива обчислювальна система протягом свого життєвого циклу пройде через низку поколінь апаратного забезпечення. Це може призвести до оновлення, оскільки обслуговування старих поколінь апаратного забезпечення може стати дорогим і непрактичним. Таким чином, планове технічне обслуговування та простої для заміни обчислювальних компонентів та компонентів зберігання даних на сучасніші версії слід розглядати як заплановану діяльність, яка може стати одним із найсерйозніших джерел недоступності системи, якщо її не вирішити за допомогою програмної архітектури, яка може забезпечити динамічну реконфігурацію критичних частин системи, тоді як решта системи залишається в режимі онлайн. Це питання планування майбутнього оновлення, розширення

та нових версій компонентів поширюється на всю складну систему, охоплюючи всі її апаратні та програмні технології.

На початку цієї роботи зазначали, що надійність програмного забезпечення найкраще розуміти як процес, що охоплює не лише відсутність у системі програмних помилок, але й такі питання, як методологія проектування програмного забезпечення, процес тестування та забезпечення якості життєвого циклу, якість механізмів самоперевірки та інтерфейсів користувача, ступінь, до якої система реалізує цільове застосування (тобто якість відповідності між специфікацією системи та специфікацією проблеми), а також механізми, що передбачені для вирішення очікуваних збоїв, обслуговування та оновлень. Це являє собою багатий, багатовимірний набір проблем, і мало які критичні системи вирішують їх так ефективно, як хотілося б. Розробники програмного забезпечення, зокрема, часто розглядають надійність програмного забезпечення спрощено, зосереджуючись виключно на специфікації програмного забезпечення, яку має реалізувати їхній код, та на його коректності відносно цієї специфікації.

Це вужче питання коректності залишається важливою проблемою; справді, багато досліджень простоїв систем у критично важливих програмах продемонстрували, що навіть після ретельного тестування програмні помилки становлять значну частку незапланованих простоїв (поширені цифри в діапазоні від 25 до 35 відсотків), і що це число надзвичайно важко зменшити. Вивчали проблеми надійності в транзакційних умовах, одного разу припустили, що залишкові програмні помилки в зрілих системах можна класифікувати на дві категорії, які вони назвали *помилками Бора* та *помилками Гейзенбага.*

Борбаг — це постійна, відтворювана проблема: якщо вона виникає, і враховуються обставини, сценарій можна відтворити, і помилка повториться. Назва покликана нагадати нам про модель атомного ядра Бора: невеликий твердий об'єкт, добре локалізований у просторі. Виявили, що в міру дозрівання систем відносна частота Борбагів з часом неухильно падає, хоча інші дослідження показують, що популяція Борбагів періодично поповнюється, коли

систему необхідно модернізувати або підтримувати протягом її життєвого циклу.

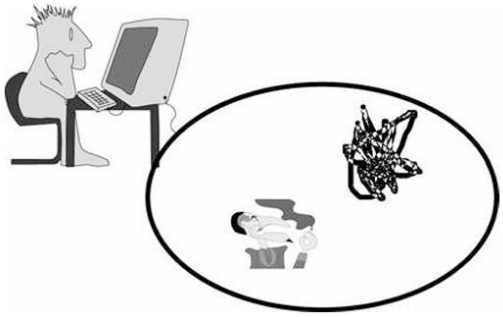

Рисунок 2.1 - Розробники, ймовірно, виявлять і виправлять «борбаги» — легко локалізовані та відтворювані джерела помилок.

Назва «помилки Гейзенберга» походить від моделі ядра Гейзенберга: складної хвильової функції, на яку впливає акт спостереження. Ці помилки зазвичай є побічними ефектами проблем, що виникли набагато раніше під час виконання, таких як переповнення масиву або випадкове розіменування вказівника після звільнення об'єкта, на який він вказує. Такі помилки можуть пошкодити програму таким чином, що це призведе до її аварійного завершення, але не раніше, ніж пошкоджена структура даних буде остаточно звернена, що може статися лише через багато часу після того, як помилка фактично була використана. Оскільки така помилка зазвичай є симптомом основної проблеми, а не прикладом самої справжньої проблеми, «помилки Гейзенберга» надзвичайно чутливі до порядку виконання. Навіть з однаковими вхідними даними програма, яка одного разу дала збій, може працювати правильно в лабораторії.

Не дивно, що основною причиною збоїв у зрілій програмній системі виявляються помилки Heisenbugs. Робота Аніти Борр йде ще далі, виявляючи, що більшість спроб виправити помилки Heisenbugs насправді погіршують ситуацію, ніж вона була спочатку. Це спостереження не дивує інженерів складних, великих програмних систем: помилки Heisenbugs відповідають

проблемам, які надзвичайно важко відстежити, і часто виправляються шляхом встановлення патчів під час виконання. Ніде розрив між теорією та практикою в надійних обчисленнях не є більш очевидним, ніж на завершальних етапах тестування та виправлення помилок великого розгортання програмного забезпечення, яке має відбуватися під тиском часу або в умовах дедлайну [35].

Кращі мови програмування можуть надзвичайно допомогти. Починаючи з появи Java і продовжуючи такими мовами, як C#, ми вперше бачимо, як велика кількість програмістів переходить до мов, які забезпечують сувору перевірку типів, автоматично обробляють збирання сміття, а також виявляють і позначають проблемні структури керування та інші можливі ознаки помилок. Обчислювальні платформи, такі як J2EE та .NET, виходять за рамки окремої програми, відстежуючи залежності від певних версій зовнішніх служб і бібліотек, і під час виконання забезпечуватимуть дотримання цих залежностей. Інструменти моніторингу та налагодження середовища виконання зробили величезні кроки вперед (раніше ми згадували, що Visual Studio .NET від Microsoft є особливо гарним прикладом, але існує багато таких систем, що виробляються десятками постачальників). Усі ці кроки дійсно можуть допомогти.

Тим не менш, не так вже й складно помилитися в Java або C# або викликати невиправний виняток під час виконання. Нескінченний цикл все одно є помилкою, незалежно від мови, а об'єктно-орієнтовані мови мають власні проблеми, такі як труднощі з простим створенням копії об'єкта. Як швидко дізнаються програмісти, знайомі з цими мовами, глибока копія створюється шляхом рекурсивного копіювання об'єкта та всіх інших об'єктів, з якими він пов'язаний, тоді як поверхнева копія зберігає пов'язані з ним об'єкти, в результаті чого до одного фізичного об'єкта тепер можна отримати доступ кількома шляхами, можливо, включаючи деякі непередбачувані. Проте створювати поверхневі копії набагато легше, і це пояснює багато помилок і помилок програмування. Нові мови не збираються усунути проблеми надійності програмного забезпечення.

Зазвичай вважається, що разом простої апаратного та програмного

забезпечення, включаючи простої для оновлення, становлять близько двох третин простоїв системи в критично важливих програмах. Решта третина простоїв пов'язана з плановим технічним обслуговуванням, таким як створення резервних копій, та факторами навколишнього середовища, такими як відключення електроенергії, несправності кондиціонерів або опалення, протікання труб та інші подібні проблеми.

Хоча надії на контроль цих форм простоїв може бути мало, тенденція полягає в тому, щоб намагатися вирішувати їх за допомогою програмних методів, які розподіляють критично важливу функціональність між достатньою кількістю комп'ютерів та розділяють їх до достатньої міри, щоб резервування могло подолати незаплановані простої . Маючи розроблене програмне забезпечення, здатне вирішувати такі проблеми, простої для обслуговування обладнання, резервного копіювання або інших рутинних цілей часто можна розглядати так само, як і інші форми простоїв. Такий підхід схильний розглядати управління системою , моніторинг та онлайн-контроль як частину самої системи: критично важлива система, по суті, повинна бути здатною моделювати власну конфігурацію та запускати відповідні дії, якщо критична функціональність з будь-якої причини скомпрометована. У наступних розділах це мотивуватиме нас розглянути проблеми, пов'язані з тим, щоб система контролювала своїх власних членів (набір процесів, що її складають) та динамічно адаптувалася скоординованим, послідовним чином, якщо виявляються зміни. Хоча потреба в стислості завадить нам розглянути питання управління системами з тим ступенем деталізації, якого заслуговує ця проблема, ми розробимо інфраструктуру, на якій можна впроваджувати надійні технології управління, і коротко розглянемо деякі нещодавні роботи, що спеціально присвячені проблемі управління.

Багато розробників стверджують, що найсерйознішою загрозою надійності розподілених систем є *складність* багатьох великих розподілених систем. Дійсно, розподілені системи, що використовуються в критично важливих програмах, часто з'єднують величезну кількість компонентів за

допомогою тонких протоколів, і результуюча архітектура може бути надзвичайно складною. Однак гарна новина полягає в тому, що коли такі системи розроблені для надійності, методи, що використовуються для підвищення їхньої надійності, також можуть протидіяти цій складності.

У наступних розділах ми розглянемо методи реплікації, які дозволяють дублювати критично важливі системні дані та служби для підвищення надійності. Ми також розглянемо новітні технології для відстеження стану системи та реагування на проблеми, якщо вони виникають під час виконання. Коли такі механізми використовуються належним чином, репліки будуть узгоджені одна з одною, і систему в цілому можна розглядати як таку, що містить лише один екземпляр реплікованого об'єкта, але такий, що є надійнішим або безпечнішим, ніж будь-який окремий об'єкт зазвичай. Якщо об'єкт активний (програма), його можна *активно реплікувати* шляхом дублювання вхідних даних та консолідації вихідних даних, які він створює. Ці методи призводять до поширення компонентів, але також нав'язують значну регулярність набору компонентів. Таким чином, вони контролюють складність, пов'язану з втручанням у стійкість.

Як щойно згадувалося, ми також розглянемо інструменти керування системою, які контролюють набори пов'язаних компонентів, розглядаючи їх як групи, в межах яких можна застосовувати спільну політику керування, моніторингу або контролю. Знову ж таки, виключаючи щось, що є істинним для всіх системних компонентів у певному класі або наборі класів, ці методи зменшують складність. Те, що раніше було набором, здавалося б, незалежних об'єктів, тепер явно розглядається як пов'язані об'єкти, які можна розглядати подібним чином, принаймні з метою керування, моніторингу або контролю.

Отже, загалом, ми побачимо, що хоча складність є серйозною загрозою для надійності, складність потенційно можна контролювати, враховуючи та використовуючи закономірності в структурі розподіленої системи — закономірності, які є поширеними, коли такі системи розроблені як керовані, відмовостійкі , безпечні або надійні в іншому сенсі. Тією мірою, якою це

зроблено, структура системи стає більш чіткою, і, отже, складність зменшується. У деяких випадках зусилля, пов'язані з побудовою системи, зростуть: цю структуру потрібно уточнити, і вона повинна залишатися точною в міру подальшого розвитку системи [36]. Але в інших випадках зусилля зменшуються: керуючи набором компонентів однаковим чином, можна уникнути необхідності робити це на ad hoc основі, що може бути подібним для членів набору, але не ідентичним, якби політики управління компонентами розроблялися незалежно.

Ці спостереження є сильною мотивацією для розгляду технологій, які можуть підтримувати групування компонентів різними способами та для різних цілей. Однак вони також вказують на другорядне міркування: якщо такі технології не будуть добре інтегровані з програмними засобами розробки систем, вони виявляться дратівливими та важкими для підтримки, оскільки система з часом розширюється. Як ми побачимо, дослідники більше займалися першою проблемою, ніж другою, але ця ситуація зараз почала змінюватися, особливо з впровадженням рішень надійності на основі CORBA, які добре інтегровані з інструментами розробки CORBA. Наприклад, CORBA пропонує архітектуру FTOL для побудови відмовостійких активних об'єктів. З іншого боку, такі функції залишаються досить попередніми і ще не прийняті «братами» (нащадками?) CORBA, J2EE та .NET. Спільнота веб-сервісів, здається, знаходиться на межі відмови від таких механізмів саме тому, що спільнота CORBA має неоднозначний досвід роботи з конкретними версіями, які вони прийняли. Жоден з цих розробок не є особливо обнадійливим для тих з нас, хто «працює» з високою гарантією.

2.2 Методи виявлення, діагностики та обробки збоїв

Дивно мало роботи було зроблено над проблемою побудови підсистем виявлення збоїв . Наслідком цього є те, що багато розподілених систем виявляють збої за допомогою тайм-аутів — схильного до помилок підходу, який змушує застосунок долати неточне виявлення збоїв у програмному забезпеченні.

Робота показує, що багато розподілених систем можуть працювати набагато краще. Фогельс проводить аналогію між виявленням збою та виявленням того, що чийсь орендар зник. Якби орендодавець намагався зв'язатися з орендарем, чек за орендну плату якого затримується, було б дещо надмірно звертатися до поліції після спроби зателефонувати цьому орендарю один раз у довільний час протягом дня та не отримати жодної відповіді. Швидше за все, орендодавець телефонував би кілька разів, розпитував би сусідів, перевіряв би, чи все ще збирають пошту, чи споживається електроенергія та вода, а також перевіряв би інші непрямі докази присутності чи відсутності орендаря.

Сучасні розподілені системи пропонують велику кількість можливостей, аналогічних цим фізичним опціям. База керуючої інформації типового обчислювального вузла (його MIB) надає інформацію про активні процеси та споживання ними ресурсів, таких як пам'ять, час обчислення та операції вводу/виводу. Часто сама мережа оснащена інструментами, і іноді можна точно виявити мережевий розділ, запитуючи MIB, пов'язані з мережевим інтерфейсом та вузлами маршрутизації. Якщо операційна система, на якій працює відповідна програма, доступна, іноді можна запитати її про стан процесів, які вона підтримує. У програмах, розроблених з урахуванням відмовостійкості, може існувати можливість інтеграції механізмів самоперевірки безпосередньо в код, щоб програма періодично перевіряла свою справність і вживала певних дій, таких як скидання лічильника, щоразу, коли перевірка успішна. Завдяки такому набору тактик можна потенційно швидко та точно виявляти більшість збоїв і навіть відрізнити збої розділення від інших збоїв, таких як збої або завершення роботи програми. Фогельс впровадив прототип служби розслідування збоїв, яка використовує ці методи, що забезпечує набагато швидше та краще виявлення збоїв, ніж традиційно вважається можливим у розподілених системах. На жаль, однак, цей підхід зовсім не є стандартним. Багато розподілених систем повністю покладаються на тайм-аути для збоїв; як і можна було очікувати, це призводить до високого рівня помилкових виявлень та значної складності подолання їх наслідків.

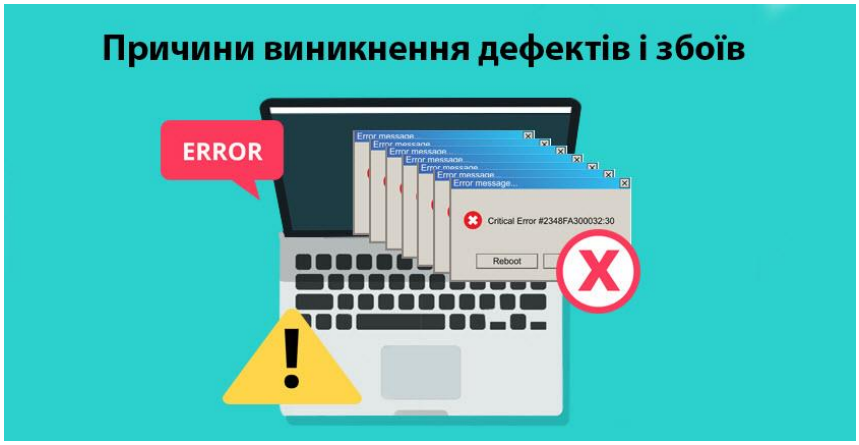

Рисунок 2.2 – Збої системи

У цьому розділі було перераховано велику різноманітність загроз надійності, які типова розподілена система може передбачати та вирішувати. Однак усі розглянуті проблеми були такого характеру, що можна вважати «рутинними» в тому сенсі, що всі вони належать до категорії створення програмного та апаратного забезпечення, стійкого до передбачуваних класів випадкових збоїв та самокерованого таким чином, щоб передбачати оновлення системи та події технічного обслуговування.

Всього кілька років тому здавалося неприродним думати про Інтернет як про вороже середовище, і таке вороже середовище постійно зростає. Сьогодні, після десятиліття вірусів та атак типу «відмова в обслуговуванні», лише дуже довірлива людина все ще вважатиме мережу безпечним місцем [37]. Сучасні комп'ютерні мережі спільно використовуються величезною кількістю комп'ютерно грамотних користувачів, чиї цілі та почуття особистої етики можуть разюче відрізнятися від цілей розробника системи. Навмисно чи ні, ці користувачі мережі становлять розсіяну загрозу; вони можуть несподівано перевірити розподілену систему на наявність слабких місць або навіть піддати її добре спланованій та організованій атаці без попереднього попередження. Навіть такі невинні програми, як особиста електронна пошта, стали мішенями для всього: від порнографічного спаму до вірусів, вбудованих в активний контент.

Спектр навмисних загроз такий же різноманітний, як і спектр випадкових загроз, розглянутий раніше. Найвідомішими із загроз є комп'ютерні віруси, які являють собою програмне забезпечення, призначене для копіювання себе з машини на машину та заподіяння шкоди машинам, на яких їм вдається закріпитися. (Доброякісний тип вірусу, який не завдає шкоди, називається черв'яком , але оскільки сама присутність непередбаченої програми може вплинути на надійність системи, можливо, найкраще дотримуватися точки зору, що всі небажані вторгнення в систему становлять загрозу для надійної поведінки.) Вірус може атакувати систему, порушуючи її припущення щодо середовища або мережі, зламуючи коди безпеки та паролі, використовуючи легітимні повідомлення або використовуючи будь-який з низки інших маршрутів. Атаки, які використовують кілька маршрутів одночасно, стають все більш поширеними, наприклад, одночасно компрометуючи певний аспект телекомунікаційної інфраструктури, від якої залежить програма, а також створюючи для програми виняткову ситуацію, яку вона може правильно обробити лише тоді, коли телекомунікаційна підсистема також функціонує.

Інші типи навмисних загроз включають неавторизованих користувачів або авторизованих користувачів, які перевищують свої звичайні обмеження. У банківській системі хтось турбується про шахрайського торговця або працівника, який намагається перенаправити кошти непомітно. Невдоволений працівник може намагатися пошкодити критично важливі системи або дані організації. У найекстремальнішому випадку можна уявити ворожі дії, спрямовані на критично важливі обчислювальні системи країни під час війни чи тероризму. Сьогодні такий вид *інформаційної війни* може здаватися підходящою темою для письменників-фантастів, проте, оскільки суспільство переносить дедалі важливішу діяльність на обчислювальні та комунікаційні технології, потенційні цілі для атаки зрештою стануть достатньо багатими, щоб зацікавити військових супротивників.

Розподілена атака типу «відмова в обслуговуванні» (DDoS) відбувається, коли група машин, часто підірваних хакерами, які проникли через мережу,

отримує завдання завалити певний сервер або центр обробки даних величезним навантаженням дороговартісних повідомлень, таких як повідомлення першої фази для встановлення TCP-з'єднання. Часто дані про походження таких повідомлень приховуються, щоб сервер не міг використовувати простий механізм фільтрації для відсіювання поганих з'єднань. Сервер виходить з ладу, і законні користувачі не можуть підключитися. Базова DDoS-атака має багато варіантів, деякі спрямовані на ключові служби, такі як DNS, деякі на маршрутизатори або «слабкі ланки» в мережі, а деякі спрямовані на інші види інфраструктури або навіть на саму програму. Більше того, DDoS-атаки можуть бути ефективними, але гарна новина полягає в тому, що вони мають обмеження.

Наприклад, коли Metallica представила свій новий веб-сайт на початку 2003 року, гурт привернув увагу спільноти музичних обмінників, яка раніше зазнавала атак з боку Metallica та її юристів через організацію під назвою RIAA. Сайт Metallica отримував близько 6 мільйонів запитів на хвилину протягом кількох днів, поки мережеві провайдери з усього світу не розробили стратегію блокування проблемних пакетів. Однак, як не дивно, досить мало шанувальників Metallica постраждали від цієї події: DDoS-трафік ускладнював реєстрацію на їхньому веб-сайті, але оскільки фактичний веб-контент кешувався у величезному центрі обробки даних, який обслуговувався Akamai, після реєстрації користувача DDoS-трафік мав незначний вплив. Таким чином, зловмисники змогли вимкнути лише один аспект усієї системи та лише на кілька днів.

Далі в цій частині роботи розглянемо питання масштабованості та побачимо, що деякі методи високої доступності мають накладні витрати, які потенційно є досить значними, коли відбуваються певні «рідкісні» події. Хитрий зловмисник може навіть розпочати атаку відмови в обслуговуванні, ініціюючи надзвичайно високу частоту таких подій, знаючи, що система заплатить за них високу ціну. Однак, оскільки ці події є нормальними, хоча зазвичай і не такими частими, системному адміністратору може бути важко навіть помітити, що виникла проблема. Наприклад, у протоколі реплікації можна неодноразово додавати, а потім видаляти певний процес — знову і знову. Протокол членства в

групі повинен запускатися щоразу, і поки це відбувається, оновлення можуть затримуватися, що фактично погіршує стан групи. Протоколи peer-to-peer часто чутливі до відтоку, аналогічної проблеми, але у великих масштабах. Таким чином, атака може навіть не ґрунтуватися на використанні якоїсь дійсно аномальної схеми трафіку!

Зрозуміло, що жодна обчислювальна система не може бути захищена від усіх можливих форм внутрішньої та зовнішньої загрози. Однак розподілені обчислення можуть запропонувати значні переваги проти добре відомого та повністю охарактеризованого профілю загроз. Розподіляючи критичну функціональність між наборами процесів, які повинні співпрацювати та координувати свої дії для виконання конфіденційних функцій, бар'єр проти зовнішніх загроз може бути непохитним [38]. Наприклад, рішення Metallica розміщувати контент на фермі веб-серверів, що обслуговується Akamai, означало, що DDoS-атаки мали б вимкнути всі тисячі комп'ютерів Akamai, щоб фактично вимкнути саму Metallica. Аналогічно , терорист, який міг би легко подолати систему, яка взагалі не має жодного захисту, зіткнувся б із набагато складнішою проблемою подолання брандмауерів, порушення меж безпеки та втручання в критичні підсистеми, призначені для продовження правильної роботи, навіть якщо обмежена кількість компонентів системи вийде з ладу або буде скомпрометована. Пізніше ми обговоримо технології віртуальних приватних мереж, які розвивають такі підходи ще далі, запобігаючи будь-якому зв'язку в мережі, крім того, що ініційовано автентифікованими користувачами. Зрозуміло, що якщо система використовує таку технологію, її буде відносно важко зламати. Однак вартість такого рішення може бути вищою, ніж може собі дозволити більшість установок.

Як розробник критично важливої системи, завдання полягає в тому, щоб передбачити загрози, які вона повинна подолати, і зробити це таким чином, щоб збалансувати витрати та вигоди. Часто профіль загроз, з яким може зіткнутися підсистема компонента, буде локалізований для цього компонента, отже, розробнику може знадобитися докласти значних зусиль для захисту деяких

особливо критичних підсистем від загроз надійності та безпеці, використовуючи при цьому набагато обмеженіші та менш дорогі технології в інших частинах тієї ж системи. Одна з цілей цієї частини роботи пов'язана з відповідним питанням — розумінням не лише того, як можна вирішити проблему надійності, але й того, як рішення можна застосувати вибірково та локалізовано, щоб розробник, який стикається з конкретною проблемою в певному контексті, міг скористатися рішенням, адаптованим до цієї проблеми та контексту, не вимагаючи реінжинірингу всієї системи для подолання вузької загрози.

Сьогодні нам бракує технології з такими характеристиками. Більшість відмовостійких та безпечних технологій вимагають, щоб розробник впровадив відмовостійку або безпечну обчислювальну та комунікаційну архітектуру, починаючи з перших рядків коду, введених у систему. За такого підходу питання відмовостійкості та безпеки стає дуже важко вирішити на пізніх етапах, коли вже існує значна кількість технологій. На жаль, однак, більшість критично важливих систем побудовані на основі вже існуючих технологій, які обов'язково будуть адаптовані до нового використання і, отже, обов'язково зіткнуться з новими типами загроз надійності та безпеці, які не передбачалися в початковому середовищі. Отже, потрібна технологічна база, яка є достатньо гнучкою, щоб навчити нас долати велику різноманітність можливих загроз, але також достатньо гнучкою, щоб її можна було використовувати вузько та вибірково (щоб витрати на надійність були локалізовані на компоненті, який робиться надійним), ефективною (щоб ці витрати були якомога нижчими) та придатною для впровадження *на пізніх етапах,* коли система вже може включати значну кількість вже існуючих технологій.

Однак, гарна новина полягає в тому, що сучасні дослідження роблять значні кроки в цьому напрямку. У наступних розділах ми розглянемо багато фундаментальних проблем, які виникають під час подолання різних класів загроз. Ми обговоримо обчислювальні моделі, які є динамічними, самокерованими та відмовостійкими, а також побачимо, як технологія, заснована на *обгортанні* вже існуючих інтерфейсів та компонентів схожими технологіями,

що впроваджують бажані функції стійкості, може бути використана для посилення складних, вже існуючих систем, хоча й з багатьма обмеженнями. Нарешті, ми розглянемо деякі проблеми великомасштабних систем, що виникають, коли складна система повинна керуватися та контролюватися в розподіленому середовищі. Хоча було б перебільшенням стверджувати, що всі проблеми вирішено, очевидно, що досягається значний прогрес у створенні інтегрованої технологічної бази для посилення критично важливих систем.

У мене мало ілюзій щодо надійності: критично важливі обчислювальні системи залишатимуться менш надійними, ніж мали б бути, доки клієнти та суспільні користувачі таких систем не почнуть вимагати надійності, а розробники не почнуть регулярно турбуватися про розуміння загроз надійності в певних умовах — планувати стратегію реагування на ці загрози та тестувати цю реакцію. Однак є підстави вважати, що в тих випадках, коли цей процес відбувається, може бути забезпечена технологічна база, здатна відповідати вимогам.

Віртуальна синхронізація підтримується деякими популярними комунікаційними пакетами, включаючи кілька, які мають бути доступні читачам цього тексту (Horus, Ensemble та Spread). Хоча засоби розробки веб-сервісів ще не забезпечують інтегрованої підтримки для таких механізмів, у майбутньому такий тип інтегрованого рішення майже напевно стане доступним. Тим часом, для тих, кому потрібна висока впевненість, існують практичні способи застосування ідей у цьому та наступному розділах до реальних систем, використовуючи стандартні засоби розробки веб-сервісів та інтегруючи їх із комунікаційними бібліотеками, що реалізують механізми, які ми вивчатимемо.

Хоча наша «справжня» мета полягає в підтримці реплікованих даних та реплікованих обчислень для високої доступності та покращеної продуктивності, ми будемо розглядати цю проблему поступово. Зокрема, основна тема, на якій ми зосередимося в цьому розділі, стосується найкращих варіантів відстеження набору членів розподіленої системи. Моніторинг членства в системі може здаватися не таким вже й центральним для реплікації даних, але насправді

відіграє фундаментальну роль: зрештою, немає сенсу говорити про реплікацію інформації, якщо ми не можемо точно пояснити, **де** мають бути репліки! Більше того, виявляється, що спосіб, яким ми вирішуємо проблему членства, має приголомшливі наслідки для продуктивності. «Правильне» визначення членства може призвести до частоти реплікованого оновлення в тисячі разів вищої, ніж у системах, які наївно підходять до проблеми членства. Дійсно, хоча багато хто вважає, що «домовленість про щось» є найфундаментальнішою проблемою розподілених обчислень, автор цього тексту міг би навести досить вагомий аргумент, що домовленість про **членство,** можливо, є справжньою суттю.

Навіщо турбуватися про домовленість щодо членства в системі? Чому б просто не довірити процесам приймати власні рішення відповідно до філософії "від початку до кінця", можливо, використовуючи тайм-аути? Читачі можуть пам'ятати, що у вступі до цієї роботи бачили, що коли тайм-аут використовується для виявлення збоїв, події, не пов'язані з несправністю, такі як перевантаження мережі, короткочасні відключення деяких комп'ютерів від мережі або зміни маршрутизації, можуть викликати тайм-аути і таким чином обдурити систему, змусивши її повірити, що такий-то вузол вийшов з ладу. Ще гірше те, що це може відбуватися вкрай суперечливими способами. Можливо, процес p зробить висновок, що процеси q, r та s працюють, але процес t вийшов з ладу, тоді як процес q вважає, що всі п'ять справні, а процес t вважає, що він єдиний, хто пережив якийсь масовий збій. Такі проблеми можуть поширитися на користувача, призводячи до заплутаної або навіть небезпечної поведінки. Наприклад, у вступі ми розглянули сценарій "розщеплення мозку", де система управління повітряним рухом може розділитися на дві паралельно розташовані системи, кожна з яких стверджує, що контролює ситуацію, але кожна не знає про іншу. Неважко зрозуміти, чому плутанина щодо членства призведе до проблем, якщо ми хочемо реплікувати дані. Припустимо, що наші п'ять процесів є частиною служби управління повітряним рухом, яка повинна підтримувати інформацію про те, які літаки знаходяться в небі та куди вони летять, і ці дані оновлюються, коли диспетчери дають інструкції літакам або коли літак змінює

свій курс. Якщо система не є послідовною щодо того, які члени працюють, вона може не оновлювати одну з реплік, і в цьому випадку ця репліка почне надавати неправильну інформацію пілотам і диспетчерам, яким не пощастить запитувати її. І навпаки, якщо ми можемо довіряти службі членства, яка повідомляє нам, які процеси «належать» системі, ми зможемо використовувати цю інформацію для підтримки простих інструментів, таких як бібліотеки, які підтримують репліковані дані та надають способи блокування елементів для ексклюзивного використання, а потім у алгоритмах вищого рівня, наприклад, для швидкого «перемикання» з процесу, який дає збій, на той, який залишається справним, щоб підтримувати майже безперервну доступність. У багатьох відношеннях угода щодо членства , таким чином, знаходиться в центрі всесвіту, принаймні, коли йдеться про обчислення з високою гарантією.

2.3 Технології обміну даними та забезпечення відмовостійкості

Було б корисно зазначити, що дані не є найважливішою частиною програми. Розробляючи програми, ми намагаємося імітувати реальні процеси, а дані є побічним продуктом. Однак дані можуть ненавмисно поєднувати частини нашої програми, і в цьому розділі представлені способи свідомого обміну

даними.

Володіння даними. Головною характеристикою, пов'язаною з даними, є право власності. Як завжди, немає чіткого правила, і все залежить від домену. Після встановлення меж важливо призначити дані базовим групам даних (БК) і лише потім проводити більш детальний розподіл даних залежно від специфіки застосунку, функціональних вимог та вимог домену.

Знову ж таки, нам потрібно розуміти, що ми моделюємо не дані, ми моделюємо поведінку, тому, наприклад, в системі інтернет-магазинів, такій як Amazon, картка покупки може бути змодельована як окрема сутність, що знаходиться в окремому контексті.

Але проблема полягає в тому, що концепція картки покупок просто

агрегує дані з інших контекстів, створюючи представлення або кеш, що, серед іншого, поєднує контекст кошика покупок з усіма іншими, що створює занадто багато складності. Ми хочемо, щоб кожен контекст представляв частину картки покупок.

Рисунок 2.3 - Картка покупки як окрема сутність (ліворуч) як концепція, розподілена між контекстами (праворуч).

Мета спільного використання визначається бізнесом.

Головна мета — зрозуміти, для чого використовуються дані. Їх можна використовувати просто для перевірки, наприклад, чи має користувач доступ, і щоб він міг його впустити. Інша причина — створення похідних даних; якщо так, то це може бути проблемою, через яку ми ділимося даними з іншими сервісами, щоб вони базували свої похідні, повертаючись до зв'язків між різними межами, пам'ятайте про свідомий обмін даними.

Деякі частини системи потребують суворої узгодженості (наприклад, переказ грошей з одного облікового запису на інший) для забезпечення інваріантності, тоді як інші можуть допускати можливу інваріантність (наприклад, поширення оновлення плану користувача) для досягнення кращої доступності або розподіленого навантаження (між працівниками) в контексті. Як описано в теоремі CAP, це або одне, або інше, ми не можемо побудувати сильно узгоджену та завжди доступну систему. Як правило, коли потрібні сильні гарантії узгодженості, частини повинні входити в один і той самий BC, ми поговоримо про причину цього в наступному розділі. Найчастіше це залежить від предметної області, наприклад, нам потрібно запитати експертів, чи

нормально, якщо користувач зможе використовувати функцію через хвилину або близько того після закінчення терміну дії його підписки. У більшості випадків це не проблема, і можлива узгодженість є достатньою [8].

Підходи до обміну даними.

Почнемо з того, що існує кілька видів даних: статичні та змінні. Статичні дані рідко оновлюються та можуть бути вбудовані в код (кожен або один сервіс, залежно від того, чи хочемо ми використовувати їх спільно, чи маємо лише один сервіс, який ними володітиме), зберігатися у спільній базі даних, у конфігурації або витягуватися в іншій MS [2].

Для змінних даних, які є найпоширенішими та найпроблематичнішими, існують такі підходи:

Переосмисліть балакучі компоненти. Перевірте, чи він активно взаємодіє з іншим сервісом. Ми б воліли перекроїти межі та об'єднати два сервіси.

Шлюз агрегації даних. Коли нам потрібно отримати дані з різних джерел (MS, BC), залежно від мети, якої ми хочемо досягти (аналітика, відображення на фронтенді) та ступеня зв'язку, який ми можемо терпіти, існують щонайменше такі підходи:

Шаблон композиції (агрегації) API – об'єднання даних за допомогою шлюзу. Він не дуже підходить для ситуацій, що вимагають об'єднання великої кількості даних (хоча для простих випадків ми можемо об'єднувати дані в пам'яті). Коли виникає проблема N+1, інші підходи, такі як CQRS, є більш підходящими.

Шаблон декомпозиції API – шлюз розділяє дані з одного запиту між кількома сервісами та збирає відповідь. Це пояснюється тим, що кожному сервісу потрібна лише частина його даних. Атомарність має бути забезпечена для таких випадків, оскільки якщо ми зіткнемося з проблемою на пізніших етапах, нам потрібно буде повернутися до всіх попередніх [2].

CQRS – техніка, яка розділяє моделі читання та запису для забезпечення кращих моделей читання для випадків використання з інтенсивними обчисленнями. Вона відрізняється від аналітичного сховища (також відомого як

OLAP, Warehouse) за призначенням, але подібна тим, що також є проекцією (кешем, представленням).

Спільна база даних.

Використовуйте одну базу даних для кількох сервісів. Це доцільно робити лише в одній базі даних, оскільки важко відстежувати, що потрібно змінити в різних MS, якщо схема бази даних не зберігається як код і не перевіряється автоматично, тому для спрощення:

- Дозволити запис лише одному MS.
- Координація змін схеми має бути присутня, тому всі сервіси повинні бути в межах однієї бази даних. Важливо зазначити, що одна команда повинна обробляти всі залежні сервіси.

Спосіб зберігання даних може не задовольняти всі випадки використання, і в такому випадку необхідно використовувати інші шаблони.

Пошук джерел для подій.

Цей підхід схожий на те, як працює система керування версіями (пам'ятайте, однак, що Git зберігає повні файли, аж до різниці), ми використовуємо сховище подій як єдине джерело достовірної інформації. Він відрізняється від підходу систем, керованих подіями, де ми зберігаємо стан і лише потім публікуємо подію. У джерелі подій ми публікуємо події та створюємо кеш зі спожитих подій, щоб пришвидшити процес запитів. Враховуючи невеликий обсяг даних, усі події можна зберігати в пам'яті.

Шаблон слід використовувати для досягнення наступних цілей.

Бізнес-причини:

- Аудит, відповідність вимогам, прозорість – потік подій надає можливість нескінченно перевіряти його на наявність закономірностей та бачити окремі кроки, які призводять до певного результату.
- Аналіз даних – оскільки у нас є потік із діями, ми можемо аналізувати дані, починаючи з самого початку, а не з моменту визначення нових вимог.
- Мати справу з часто змінюваними вимогами або невідомими майбутніми вимогами [17].

Технічні причини:

- Гарантована повнота виявлених подій – після публікації подію неможливо видалити. Дозволено лише опублікувати нову подію, щоб скасувати зміни, внесені попередньою.
- Єдине джерело достовірної інформації – сховище подій тепер містить всю інформацію, необхідну для того, щоб новий сервіс міг її отримувати.
- Сприяє налагодженню — маючи послідовність подій, можна переглянути їх, щоб побачити, звідки виникла проблема та що її спричинило.
- Відтворення в нові моделі зчитування (CQRS) – потік подій легко конвертується в кеш для різних цілей.
- Мати справу зі складністю моделей – пошук джерел подій змушує нас дотримуватися вузькофокусованих моделей, які легше зрозуміти.

Збір даних про зміни (CDC).

Синхронне оновлення кешів робить застосунок менш стійким та доступним. Наприклад, якщо ми оновлюємо індекс пошуку або кеш Redis під час кожного оновлення продукту, що станеться, якщо один із цих компонентів буде недоступний? Чи пропустимо ми оновлення, чи чекатимемо, поки компонент відповість? Це не означає, що ми ніколи не повинні оновлювати зовнішні системи в межах початкової транзакції через можливість невідновлення після збою; ми не зможемо забезпечити атомарність у DS, про що ми поговоримо в наступному розділі.

У таких випадках було б зручно використовувати події для поширення оновлень даних. Події у випадку джерела подій дозволяють споживачеві зрозуміти їх, і підхід не такий простий. У випадку CDC ми зазвичай працюємо лише з фіктивними подіями двох типів: upsert та delete, крім того, ми не обираємо сценарій єдиного джерела достовірності, використовуючи такі методи, як ведення журналу транзакцій БД або інші, тому БД залишається основним джерелом.

2.4 Висновок по розділу

У другому розділі було досліджено методи та інструменти забезпечення надійності розподілених систем. Проаналізовано основні причини виникнення відмов, збоїв та помилок у розподілених обчислювальних середовищах. Розглянуто сучасні методи виявлення, діагностики та обробки несправностей, які дозволяють своєчасно локалізувати проблеми та мінімізувати їх вплив на роботу системи. Також досліджено технології обміну даними та механізми забезпечення відмовостійкості, включаючи резервування, реплікацію та балансування навантаження. Отримані результати підтверджують, що комплексне використання зазначених підходів сприяє підвищенню стабільності та безперервності функціонування розподілених програмних систем.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ МЕТОДІВ ЗАБЕЗПЕЧЕННЯ НАДІЙНОСТІ

3.1 Платформні технології для побудови надійних розподілених систем

Низка компаній конкурує, пропонуючи комплексні, інтегровані платформи, за допомогою яких розробники можуть легко створювати розподілені обчислювальні програми. Серед лідерів – Microsoft зі своєю системою .NET та Sun з платформою J2EE Java. Це не єдині гравці, навіть якщо уявити собі їх якомога ширше: компанії з управління базами даних та інформацією, такі як Oracle, SAP, BEA та PeopleSoft, пропонують власні платформи. IBM та HP мають широкі технологічні пропозиції, які лідирують на своїх відповідних ринках, і цей список можна продовжувати. Тим не менш, .NET, ймовірно, є найпомітнішою платформою для настільних та персональних комп'ютерів, тоді як J2EE, ймовірно, є найвідомішою платформою для використання на серверах різних типів, витіснивши CORBA. З цієї причини, а також заради стислості, цей розділ зосереджується лише на цих двох платформах, а також пропонує їх коротке порівняння.

Переходячи до суті, можна виявити, що такі системи, як .NET та J2EE, багато в чому досить схожі. Обидві пропонують широкий функціонал, спрямований на програми, що здійснюють доступ до баз даних, обидві мають дуже потужні рішення для веб-інтеграції та обидві мають комплексну підтримку широкого спектру програм загального призначення. Хоча J2EE розроблявся з акцентом на Java, а .NET - на нову мову під назвою C# (і другу під назвою J#, що використовує синтаксис Java), жодна з них на момент написання цієї роботи не є насправді «мовно-орієнтованою». Зрештою, ці дві системи відрізняються саме їхнім фокусом. J2EE створювався з надією, що Java та J2EE можуть витіснити інші форми мов корпоративних обчислень, і був напрочуд ефективним у цьому. Це змушує багатьох розробників J2EE думати про рішення для своїх програм на

чистому Java. На противагу цьому, .NET був представлений, коли J2EE вже давно був на ринку, і Microsoft позиціонував платформу, щоб підкреслити її інтеграцію та міжмовні можливості. Результатом цього стало те, що на момент першої пропозиції .NET багато клієнтів одразу ж перейшли на нього, розчаровані труднощами, з якими вони зіткнулися під час інтеграції програмного забезпечення Java зі застарілими додатками. Протягом року J2EE запропонувала вдосконалену технологію «адаптера об'єктів», яка, здається, зрівняла умови гри. Однак навіть після цього J2EE продовжує мати вигляд одномовної платформи, хоча б тому, що документація здебільшого передує цьому незначному зрушенню.

Однак .NET явно надає перевагу чистому середовищу Microsoft, і хоча існують способи експорту технології .NET для використання на платформах, відмінних від .NET, вони дещо обмежені та створюють неприродне відчуття для типового розробника Linux.

Сьогодні, звичайно, обидві спільноти здебільшого зосереджені на веб-сервісах, і боротьба за першість на ринку, що розвивається, ще далеко не вирішена; справді, інші гравці, такі як BEA, IBM або HP, цілком можуть стати домінуючими, щойно ситуація нарешті вщухне. Кінцеві користувачі та розробники оцінюють ці платформи за багатьма параметрами: простота розробки додатків, сумісність, стабільність і безпека, загальна вартість володіння (тобто вартість управління платформою після її розгортання) тощо [39]. Сама ідея використання єдиного рішення в багатьох аспектах сумнівна, оскільки головна мета таких платформ полягає у сприянні сумісності між існуючими, досить різноманітними платформами та застарілими додатками.

Таким чином, у цьому розділі ми не маємо на меті детальний огляд двох найкращих у своєму класі рішень і не маємо на меті рекомендувати одне над іншим. Швидше, наша мета - розглянути, як два основні постачальники відреагували на сприйнятий ринковий тиск, включивши ті механізми, про які ми читали, у свої флагманські лінійки продуктів.

.NET Framework – це інфраструктура для всієї платформи .NET. Вона

включає бібліотеки базових класів, такі як ADO.NET та ASP.NET, а також середовище виконання Common Language Runtime. Фреймворк забезпечує базову функціональність, доступну для всіх .NET-застосунків – його можна уявити як операційну систему та пов'язані з нею бібліотеки, об'єднані в єдиний пакет та інтегровані з (великою) різноманітністю стандартних серверів та служб, що знаходяться в Інтернеті в цілому або виключно у світі Windows.

Microsoft пропагує використання низки мов програмування та дедалі більше схиляється до розгляду всіх систем як збірок об'єктів, де мова, що реалізує об'єкт, є вибором розробника. Таким чином, .NET пропагує нову мову під назвою C# (вона схожа на Java, але з деякими синтаксичними відмінностями, здебільшого дуже незначними, та багатьма відмінностями від пакета середовища виконання), але також підтримує величезну кількість інших мов, включаючи J# (варіант Java від Microsoft з ідентичним синтаксисом, але з використанням бібліотек C#), Visual Basic (напрочуд об'єктно-орієнтована версія мови Basic), C++, C, Cobol, SmallTalk, Pascal, ML тощо. Особисто я чекаю на відродження Snobol, але тим часом я почав працювати на C#. Кілька з цих мов надзвичайно добре підтримуються провідним середовищем розробки додатків Microsoft, Visual Studio.

Як згадувалося раніше, найважливіші цільові спільноти для .NET, як правило, потребують надзвичайно чистого доступу до систем баз даних і зосереджені на створенні веб-сервісів або веб-інтерфейсів для своїх програм. ADO.NET надає доступ до бази даних програмісту .NET через дуже просте, чисте віртуальне середовище бази даних, яке можна підключити до «реальної» бази даних на вибір. ASP.NET надає простий спосіб перетворити .NET-застосунок на систему веб-сервісів або веб-сервер для більш традиційного управління документами. ASP.NET також дозволяє легко надсилати запити до віддаленої системи веб-сервісів з .NET-застосунку.

CLR, або Common Language Runtime, забезпечує середовище виконання для .NET-застосунків, незалежно від їхньої мови програмування (це означає, наприклад, що в .NET можна викликати функції безпосередньо з C# у Visual

Basic, а звідти в COBOL, що полегшує взаємодію між новими застосунками та застарілими системами). CLR забезпечує можливість компіляції «точно вчасно» (ідея полягає в тому, що мови компілюються в проміжну мову програмування, яка виглядає як машинний код, але насправді є платформонезалежною, а потім ця мова, у свою чергу, компілюється в справжній машинний код в останній момент), управління пам'яттю та збирання сміття (мені ніколи не подобалося збирання сміття через непередбачувані витрати, але така можливість, безумовно, зменшує кількість помилок). CLR також розуміє збірки та може фактично відстежувати та запускати кілька паралельних версій бібліотек, коли, як це часто буває, потрібно запустити два застосунки, і вони базуються на різних версіях важливої системної бібліотеки. Традиційно завантаження новішого застосунку призводило до оновлення бібліотеки та ризику спричинити проблеми для старішого. З підходом Microsoft виникають проблеми, якщо сама програма має якусь базу даних і не може підтримувати кілька версій свого коду пліч-о-пліч, але ризик того, що оновлення порушить роботу старіших програм, принаймні мінімізований.

Рисунок 3.1 – Середовище C# .NET

Мабуть, найбільшою силою платформи .NET є її середовище розробки Visual Studio, яке також інтегровано з навчальними посібниками та величезною кількістю онлайн-довідки для всього спектру підтримуваних мов і технологій, включаючи приклади копіювання та вставки. Хоча це не робить розробку додатків тривіальною, це може допомогти новому користувачеві подолати складні моменти та значно підвищити продуктивність. Крім того, Visual Studio

має потужні інтегровані інструменти налагодження та аналізу виконання. Незважаючи на відсутність механізмів аналізу продуктивності та технологій для забезпечення високої надійності, пакет Visual Studio .NET спрямовує розробників до рішень, які, ймовірно, матимуть хороші властивості надійності та будуть природно поєднуватися (а отже, добре підтримуватися) з іншими елементами платформи .NET. Звичайно, ця проста інтеграція не поширюється на платформи, відмінні від Windows, та «стандартну» Java, яку Microsoft підтримує, але не просуває. Проте факт залишається фактом: Visual Studio .NET є надзвичайно привабливим входженням у бачення продуктів Microsoft та прикладом для інших компаній, що показує, як дуже складні нові технології можна відносно безболісно впроваджувати для дуже широкої аудиторії.

3.2 Багаторівнева архітектура та організація взаємодії процесів

Відмовостійкі методи

Механізм відмовостійкості працює як основа розподілених систем і є одним із важливих типів методів життєвого циклу відмов [38] , які відіграють важливу роль у надійності розподілених корпоративних застосунків. Він має справу з усіма тими ситуаціями, коли виникають збої або несправності під час виконання корпоративного застосунку в реальному часі, і забезпечує можливість системи продовжувати коректно виконувати свої функції. Виходячи з різних непередбачених ситуацій у розподіленому середовищі, такому як сервісно-орієнтована архітектура, Grid та хмарне середовище, збої або несправності можна класифікувати на три категорії [40].

i. е. Рівень програми, системний рівень та мережевий рівень. Для своєчасної обробки цих помилок відмовостійка система може використовуватися як просторова або часова резервування, включаючи реплікацію апаратного забезпечення (з додатковими компонентами), програмного забезпечення (зі спеціальними програмами) та часу (з диверсифікованими операціями) [6] . Реплікація з точки зору даних має більший

вплив на продуктивність великомасштабних розподілених систем. [45,28] запропонували оптимальне рішення для прозорої реплікації даних у розподіленому середовищі, коли дані складаються з мультимедійних об'єктів. Зі значним прогресом у галузі розподілених обчислень, розробка динамічних розподілених корпоративних програм значно зросла завдяки гнучкій архітектурі. Були впроваджені різні методи відмовостійкості, щоб забезпечити надійне та безперебійне уявлення для кінцевого користувача. Нижче ми навели аналіз різних моделей/стратегій, що забезпечують відмовостійкість у різних обчислювальних середовищах. Ми також підсумували можливі недоліки або відкриті питання в цих методах.

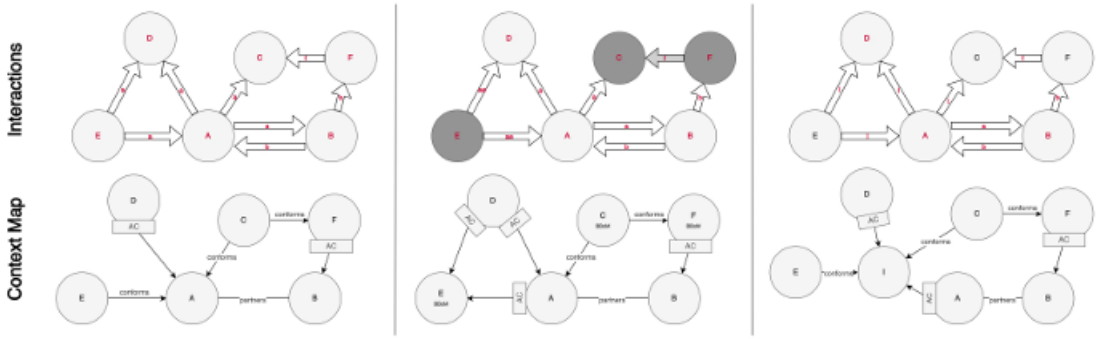

Рисунок 3.2 - Різниця між картою контексту та фізичними взаємодіями мовних спільнот (перший); поширення помилок у динамічній системі (другий); створення проксі-мовної спільноти для уможливлення мовної зміни (третій)

Більшість цих методів просто перенаправляють запити клієнта на резервні сервери у разі будь-якої помилки, не враховуючи поточний процес виконання. Запити, що виконувалися, фактично відкидаються, і не було визначено жодного механізму для відновлення цих транзакцій для безперебійної взаємодії. Однак [9] визначили механізм реєстрації для моніторингу стану виконуваних процесів та запропонували модифікацію ядра Linux та веб-сервера Apache для забезпечення правильної обробки запитів у процесі під час збою. Однак це має свій недолік, як зазначено в таблиці 4.

Методи прогнозування несправностей або збоїв

Ми проаналізували кілька моделей з точки зору різних факторів,, для прогнозування або вимірювання надійності розподілених систем, які можна приблизно класифікувати на моделі, орієнтовані на користувача, моделі, засновані на архітектурі, та моделі, засновані на стані. Наш аналіз показує, що майже всі моделі, засновані на цих підходах, враховують різні фактори, такі як різні умови експлуатації - високий рівень використання та перевантаження, збої обладнання - (жорсткі диски, мережеве обладнання, сервери, джерела живлення, пам'ять, процесор), взаємодія із зовнішніми службами або програмами, випадкові події - збої безпеки та проблеми, пов'язані з операційним середовищем, при прогнозуванні або вимірюванні надійності розподілених систем. Однак майже всі ці моделі мають два недоліки. 1) Вони не враховували всі фактори разом або вважали деякі з них постійними. 2) Вони вважали збій обладнання важливим фактором постійним або ігнорували його при вимірюванні або прогнозуванні надійності програмного застосунку, як показано на рисунку 3.3 .

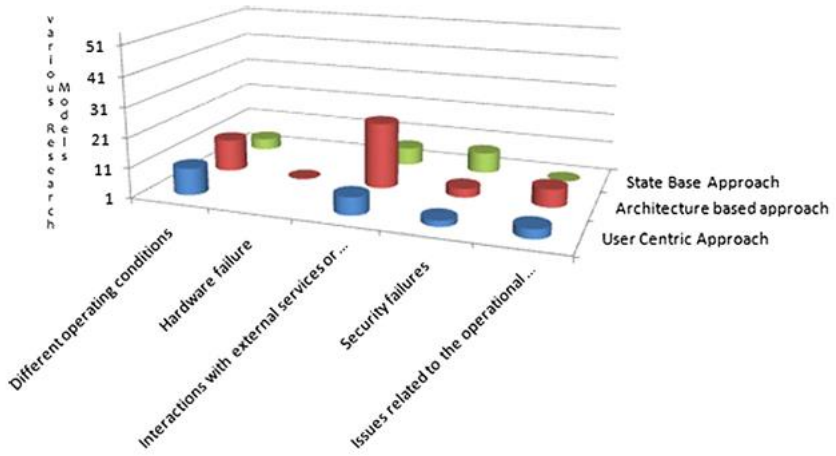

Рисунок 3.3 Графік, що ілюструє використання різних факторів у різних моделях прогнозування надійності.

Користувацько-орієнтовані підходи можна охарактеризувати як багатоетапні процеси вирішення проблем, де система розглядається з точки зору поведінки користувача. Оскільки надійність будь-якої системи має прямий вплив

на використання системи, ці моделі прогнозують надійність, враховуючи всі показники для обох типів зацікавлених сторін, тобто користувачів послуг та постачальників послуг. Оцінка надійності великомасштабних систем на архітектурному рівні є складним завданням, оскільки без наявності операційного профілю або профілю використання важко знати реальний вплив будь-якої послуги (послуг) до її розробки [2]. Тому надійність таких програмних систем значною мірою залежить від операційного профілю або профілю використання заданого набору послуг. Модель, що базується на часі, в основному працює за принципом оцінки часу передачі для обчислення часу виконання кожного файлу або програми в реальних умовах роботи в розподіленому середовищі [25] . Але оскільки системи поводяться по-різному за різних обставин, різні фактори, такі як непередбачувана швидкість Інтернету, мережева інфраструктура зі змінними каналами зв'язку, інтуїтивно впливають на надійність одного й того ж набору послуг для різних користувачів послуг за різних обставин [3] .

Існує багато варіацій у прогнозуванні надійності продуктивності за моделями, що базуються на часі, та моделями прогнозування на рівні архітектури. У цьому відношенні зроблені деякі інші вимірювання, які можна назвати орієнтованими на користувача підходами, такими як [19,26] та системи на основі сітки [17] . Ці підходи пояснюють різні проблеми надійності як з точки зору кінцевого користувача, так і з точки зору постачальника послуг, та обговорюють, як склад сервісів або середовище сітки впливає на надійність додатків. Модель WS-DREAM [26] базується на механізмі спільної оцінки для оцінки надійності в SOA-додатках шляхом надання тестового середовища в режимі реального часу для тестування надійності сервісів різним користувачам у різних географічно розподілених місцях. Тут усі користувачі можуть ділитися своїми результатами на центральному сервері, що полегшує оцінку бажаного сервісу в розподіленому середовищі. Цей тип підходу зазвичай називається підходом тестування чорної скриньки, де користувачі стурбовані лише кінцевим результатом. Оскільки вони надають лише тестове середовище через Інтернет, ігноруючи різні важливі фактори, такі як невизначеність у каналі зв'язку та

змінна сукупна кількість користувачів або різна інфраструктура в центрі обробки даних. Таким чином, його не можна порівнювати з середовищем реального часу для екстраполяції надійності сервісів. Відповідно до моделі WS-DREAM, у роботі [19] також представлено орієнтований на користувача підхід для розподіленого застосунку на основі SOA та виділено подвійну поведінку сервісів для різних користувачів, а також доведено, що надійність розподіленого застосунку на основі SOA знижується з точки зору постачальника зі збільшенням кількості сервісів у складі, проте зростає, якщо розглядати ту саму структуру та аналізувати її з точки зору кінцевого користувача.

Грід-середовище базується на багаторівневій структурі з гетерогенних компонентів [17]. Кожен рівень працює під власною юрисдикцією, наприклад, Grid Fabric зазвичай має справу з робочими вузлами, операційними системами, локальними планувальниками або мережевими зв'язками, Core Grid Middle Ware займається подачею завдань, виявленням та моніторингом ресурсів або службами управління даними, а User Level Grid Middle Ware відповідає за метапланувальники, брокери ресурсів або Грід-портали. [17] представили процедуру оцінки надійності грід-системи та зосередилися лише на User Level Grid Middle Ware з точки зору користувача. Згідно з їхнім аналізом, розумний рівень надійності для кінцевого користувача може бути досягнутий за допомогою метапланувальника Grid-Way на будь-якій інфраструктурі на базі Globus.

Підходи, засновані на архітектурі

Прогнозування надійності на етапі проектування в сервісно-орієнтованій архітектурі (SOA) [1-3,21] має не тільки більший вплив на якість програмного забезпечення, але й допомагає зменшити витрати на реінжиніринг, забезпечуючи більш надійні програмні додатки. У випадку сервісно-орієнтованої архітектури [3] використовувався підхід спільного використання даних про відмови між користувачами для прогнозування надійності для подібного набору користувачів та послуг на основі їхнього минулого досвіду. Для пошуку подібного набору користувачів та послуг використовується коефіцієнт кореляції Пірсона (PCC),

який вже рекомендувався різними системами. Цей підхід здається досить переконливим, але все ж має деякі технічні проблеми, оскільки, окрім подібних послуг та подібних користувачів послуг, подібна інфраструктура також може бути критичним аспектом у визначенні показників надійності на основі старих даних. Оскільки поведінка системи представлена низькорівневою функціональністю системи, надійність низькорівневої функціональності системи сприятиме загальній надійності системи [18] . Тому подібного досвіду користувачів для подібного набору послуг недостатньо для прогнозування надійності для подібного набору послуг.

Багато дослідників дотримуються думки, що постачальники послуг повинні надавати деякі інші деталі для обчислення надійності обох типів послуг, тобто атомарних послуг та композитних послуг. Наприклад, зовнішні служби, які вони використовують, як служби об'єднані в композитній службі, як часто вони викликають одна одну, та графік потоку, що описує поведінку служби [2] . Сервісно-орієнтована модель надійності (SORM) [18] обчислювала надійність атомарних та композитних послуг, використовуючи різні методи. Вона використовувала високоефективне групове тестування для оцінки надійності атомарної служби на основі голосування більшості та оцінювала надійність композитної служби за допомогою моделі на основі архітектури, надійності кожної атомарної служби, сценаріїв виконання та операційних профілів.

У розподілених застосунках на основі *сервісно-орієнтованої архітектури (SOA)* сервіси можуть належати та розміщуватися різними організаціями. Розробка розподілених систем шляхом інтеграції цих сервісів та прогнозування або аналізу надійності таких систем є складним завданням. Методологія прогнозування надійності, представлена в [1], аналізує вплив на надійність таких сервісів, коли вони збираються шляхом:

1. Чітке розуміння залежності між вхідними параметрами сервісів та вхідними параметрами каскадних сервісів.
2. Вплив спільного використання послуг.

Однією з переваг цього підходу над іншими є використання характеристик як високорівневих, так і низькорівневих сервісів. Він має справу не лише зі зібраними сервісами, але й враховує сервіси, що пропонуються програмним компонентом, та сервіси, що пропонуються комунікаційними пристроями. Однак вони припускали, що часткова інформація про надійність віддаленого веб-сервісу має бути опублікована з кожним сервісом, що є найважливішим аспектом у застосунках на основі SOA.

ATOP (діаграма активності для PRISM) [21] – це ймовірнісний підхід, що керується моделлю, який також підтримує ранню оцінку якості обслуговування складу послуг на архітектурному рівні під час фази проектування та автоматично перетворює проектну модель складу послуг на аналітичну модель, яка потім подається до засобу перевірки ймовірнісних моделей, такого як PRISM, для перевірки надійності. Він починається з високорівневого опису складу послуг, який автоматично генерує випадкову модель, яку можна розв'язати за допомогою різних функцій засобу перевірки моделей PRISM. Однак ця модель сильно залежить від результатів

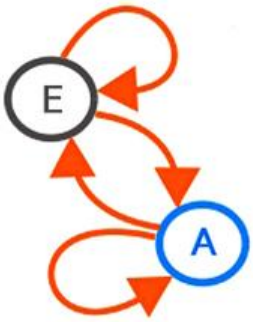

Рисунок 3.4 - Процес на основі ланцюга Маркова

Ймовірнісна модель PRISM, що використовується для перевірки надійності зовнішніх сервісів та специфікації зовнішнього сервісу при використанні в композиції, тобто їх функціональних та нефункціональних атрибутів.

Підходи, що базуються на державі

Відомий ланцюговий процес Маркова – це дискретний випадковий процес, який переходить з одного стану в інший, як показано на рис. 4,

ланцюгоподібним чином [30] для відображення шарів у різні фізичні стани, де кожен наступний стан залежить від поточного стану, а не від усієї системи. Різні моделі використовували ланцюговий процес Маркова для аналізу або прогнозування надійності розподілених програм шляхом обчислення часових рамок для кожного виконуваного завдання, щоб визначити, чи успішно вибране завдання виконало своє завдання у задані часові рамки чи ні.

Надійність усієї програми дуже залежить від надійності кожного її окремого компонента, оскільки відмова будь-якого компонента впливає на надійність інших. Ігнорування будь-яких артефактів програми під час прогнозування надійності може бути фатальним; будь-яка окрема відмова може стати причиною відмови сервісу. Для обробки всіляких помилок, таких як збої через тайм-аут, блокування, збої мережі тощо, які можуть виникнути під час певної операції виклику сервісу [23], описано ієрархічну модель. Ця ієрархічна модель пропонує вирішувати різні помилки на різних рівнях та використовує принцип станів Маркова для відображення рівнів у різні фізичні стани. Виникнення збоїв під час певної операції робить сервіси мережі ненадійними [23] . Ця модель використовує моделі Маркова, теорію черг, теорію графів та байєсівський аналіз для прогнозування надійності мережі шляхом обробки блокування, тайм-ауту, підбору пар, збоїв мережі, програми та ресурсів ієрархічним чином. Помилки, що виникають на різних рівнях, можна передбачити, вказавши певні заздалегідь визначені критерії, такі як, у випадку рівня запитів, у будь-якому стані кількість запитів, що перевищує довжину черги запитів, призводить до переповнення запитів, і згідно з цією моделлю, проблема сталої ймовірності перебування системи в цьому стані може бути розв'язана за допомогою рівняння Чепмена-Колмогорова. Системи з обмеженнями в часі є критичними, оскільки вони повинні виконати призначене їм завдання або роботу у визначені часові рамки. Розрахунок часових рамок для кожного виконуваного завдання, мережевого часу та часу обробки завдання є ключовими показниками, які можна використовувати. Модель на основі часу передачі [25] також працює за тим самим принципом оцінки часу передачі для обчислення часу виконання

кожного файлу або програми в реальних умовах роботи в розподіленому середовищі, потім на основі цієї інформації про обмеження в часі та часу виконання кожної програми або файлу в розподіленому середовищі можна визначити відповідний марковський стан для обчислення надійності.

Час зв'язку та час обробки в середовищі грід-обчислень є важливими факторами для аналізу надійності грід-системи. Під час передачі даних між двома різними вузлами в грід-середовищі збій вважатиметься невдалим, якщо станеться будь-який збій або на вузлах, або в каналі зв'язку між цими вузлами. Аналогічно, програма, що виконується на певному вузлі, вважатиметься невдалою, якщо станеться будь-який збій на цьому вузлі. Алгоритм, визначений для генерації MFST [25], не може бути використаний для обчислення мінімального охоплюючого дерева ресурсів (MRST), оскільки він не враховує один фактор, тобто час роботи [24] . Модель, запропонована в [24], шукає MRST, враховуючи всі задіяні фактори, такі як ресурс, елемент, час роботи, вузол та канал зв'язку. Вона базується на механізмі умовної ймовірності для обчислення надійності грід-програми, тобто MRST-1 не працює за умови, що MRST-2 працює, і пропонує ймовірність перетинів набору MRST кожної програми, оскільки будь-який з перетинів MRST може гарантувати працездатність усіх обчислювальних програм у системі грід-обчислень.

Хоча в системах грід-обчислень вже досягнуто значного прогресу, масштабовані рішення для забезпечення надійності грід-інфраструктури ще не вирішені. Різні моделі, розроблені для прогнозування надійності функціональних областей у грід- ресурсах, в основному призначені для розробки методів для відмовостійкої системи, тобто виявлення несправностей та збоїв у грід-ресурсах до їх впровадження та відновлення, щоб дозволити обчисленням продовжуватися без збоїв [16] . Певні обмеження в існуючих моделях аналізуються з урахуванням вищезазначених функціональних областей виявлення несправностей у розподілених середовищах, наприклад: механізм виявлення несправностей, розроблений для грід-систем, не вважається придатним для великомасштабних, гетерогенних , динамічних грід-систем [16],

так само групі розподілених детекторів відмов неможливо досягти консенсусу щодо того, які ресурси вийшли з ладу, чи вийшов з ладу будь-який компонент, що бере участь у процесі обчислення [31] тощо. [16] виділено та запропоновано деякі ключові специфікації, які потребують подальшого детального аналізу, щоб зробити грід-систему більш надійною або відмовостійкою. Ключові специфікації включають архітектурний підхід, кількісний підхід та виявлення поведінки окремих компонентів.

Після інтенсивних досліджень у галузі віртуалізації, грід-обчислень та сервісно-орієнтованої архітектури (SOA) була розроблена концепція хмарних обчислень. Аналіз/моделювання надійності та показники продуктивності розподілених хмарних додатків є складними, оскільки ці додатки не відповідають додаткам розподілених обчислювальних систем або грід- та SOA-додатків з точки зору їхнього складу , конфігурації та вимог до розгортання. Окрім різних проблем у хмарних системах, наприклад:

1. Як збирати інформацію про походження стандартизованим та безперешкодним способом.
2. Як представити цю інформацію користувачеві логічно, тобто через інтуїтивно зрозумілий веб-інтерфейс користувача [32] .

Існують і інші фактори, такі як надійність та доступність різних хмарних компонентів і сервісів. Відмова будь-якого ресурсу в хмарній системі може призвести до збою всієї програми, що працює в цьому середовищі. У роботі [33] було представлено детальний аналіз характеристик апаратних відмов, а також попередній аналіз предикторів апаратних відмов. У роботі [34] було представлено модель оцінки довіри для ефективної реконфігурації та розподілу різних хмарних ресурсів кільком користувачам для їхніх численних запитів. Модель оцінки довіри базується на історичній інформації, зібраній з журналів кількох серверів у хмарному середовищі, для прогнозування найкращих доступних ресурсів залежно від запитів різних користувачів у майбутньому, що може допомогти постачальникам хмарних послуг ефективно використовувати

певні хмарні ресурси, прогнозуючи невизначену поведінку запитів кількох користувачів.

У хмарному середовищі неможливо просто використовувати лише одну модель для прогнозування надійності програмного забезпечення, такої як надійність мережі, надійність апаратного забезпечення, надійність програмного забезпечення, оскільки вони взаємопов'язані одна з одною, одна може спричинити збій іншої, впливаючи на загальну надійність системи [22] . Для системи, що працює з менш надійними компонентами, надійність стає проблемою продуктивності, оскільки надійність та методи відновлення впливають на продуктивність [16] . [22] змоделювали двоетапний підхід, тобто етап запиту та етап виконання, для прогнозування надійності хмарних систем. Оскільки надійність певним чином відноситься до успішного виконання послуги навіть за наявності несправностей, цей двоетапний підхід класифікує різні помилки на відповідні етапи залежно від їхньої природи, наприклад, ті, що пов'язані зі зв'язком, базою даних, апаратним або програмним забезпеченням тощо, обробляються на етапі виконання, а інші, такі як помилки тайм-ауту або переповнення, обробляються на етапі запиту. Щоб хмарний сервіс був надійним, як етап запиту, так і етап виконання повинні успішно завершити бажану послугу у певний термін за різних обставин.

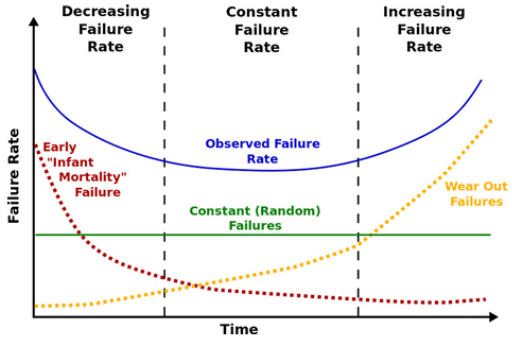

Рисунок 3.5 - Приклад кривої «ванни», що ілюструє характерну особливість виходу з ладу апаратного забезпечення.

Зв'язок

Наступний важливий момент, окрім того, чим варто поділитися, — це підхід. Шаблони комунікаційного проектування керують тим, як ми будуємо зв'язки між компонентами, і обираються на основі багатьох факторів.

Стратегія.

Стратегія зв'язку може бути синхронною або асинхронною, що відрізняється від протоколу, такого як HTTP або AMQP, це радше концепція, пов'язана зі зв'язком, коли ми не викликаємо інші MS в межах початкового запиту. Ми використовуємо події або інші засоби для відкладення цих дій. «Підхід на основі HTTP цілком прийнятний; проблема тут пов'язана з тим, як ви його використовуєте. Якщо ви використовуєте HTTP-запити та відповіді лише для взаємодії зі своїми MS з клієнтських програм або шлюзів API, це нормально. Якщо ви створюєте довгі ланцюжки синхронних HTTP-викликів між MS, спілкуючись через їхні межі так, ніби MS є об'єктами в монолітному застосунку, ваш застосунок зрештою зіткнеться з проблемами продуктивності, зв'язку та поширення збоїв» [4].

Мета комунікації стосується того, чи хочемо ми запитувати деякі дані, чи змінювати їх, а також мети комунікації.

Якщо нам важливо, щоб умова була істинною пізніше після початкового запиту, то, по суті, ми хочемо зарезервувати деякі ресурси і тому прагнемо досягти атомарності в DS. Те саме стосується серії мутацій, які повинні бути успішними або повністю, або жодної.

Атомність у DS.

Атомарність, перш за все, визначається доменом. Чи потрібна нам атомарна операція? Чи нас цікавить скасування змін, коли викликається кілька сервісів, і ми виходимо з ладу посередині, а дані впливають на ці інші сервіси? Чи нас цікавить скасування змін, коли ми оновлюємо два різні документи в базі даних NoSQL (БД) і виходимо з ладу перед оновленням другого? Ми завжди повинні припускати, що якщо проблеми можуть виникнути, вони врешті-решт виникнуть, то чи буде бізнес терпіти це?

Існує залежність між ресурсом та межею транзакції-атомності, на яку впливає те, як ми толеруємо часткову недоступність системи (згідно з теоремою CAP) та тип системи.

- У базах даних SQL, які підтримують транзакції ACID, все є транзакційним; однак вони погано масштабуються.
- Окремий написаний документ є атомарним у NoSQL базах даних, які підтримують принципи BASE. І навіть якщо транзакції доступні в деяких базах даних, таких як MongoDB, вони працюють погано і є радше винятком, ніж нормою. Однак це не означає, що атомарно оновлювати кілька документів неможливо, просто потрібен інший підхід [9].
- У різних ресурсах, таких як база даних та черга, немає транзакційності. Для цього ми використовуємо шаблон під назвою вихідні повідомлення, який зберігає всі опубліковані події в одній транзакції, а запис у базу даних потім обробляється та публікується [14].

Отже, коли ми говоримо про зв'язок між різними MS, атомарність відсутня, якщо ми не використовуємо один із підходів: двофазні коміти або шаблон Saga. Ці підходи різні залежно від стратегії зв'язку, де синхронний режим використовується двофазними коммітами, а асинхронний - шаблоном саги.

Двофазні коміти.

Це, мабуть, перший підхід, який спадає на думку, коли потрібно забезпечити атомарність операцій на кількох MS: давайте створимо довготривалу транзакцію в кожному сервісі, з яким ми взаємодіємо, та оновимо дані, але зафіксуємо транзакцію лише тоді, коли отримаємо другий запит на підтвердження. Багато чого може піти не так, особливо якщо є проблеми з мережею. Більше того, цей підхід вимагає, щоб усі сервіси були запущені [15].

Візерунок саги.

Порівняно з попереднім підходом, замість того, щоб уміщати все в одні й ті ж межі транзакції, ми приймаємо остаточну узгодженість. Ідея полягає в тому, щоб розбити весь процес на етапи, які запускаються або скасовуються за

допомогою повідомлення. У разі збою одного етапу, всі інші мають бути скасоване шляхом прослуховування та обробки компенсуючої події [21].

Під час роботи із сагами спостережуваність необхідна через складність комунікації між багатьма компонентами. Ще однією проблемою є необхідність впровадження та тестування компенсуючих операцій, що не так просто. Доки сага не буде зафіксована, дані можуть бути зчитовані некоректно; у різних контекстах може виникнути потреба також зчитувати некоректну інформацію, наприклад, у банківській справі зазвичай є чистий та некоректний баланс рахунку, де перший визначається всіма завершеними транзакціями, а другий також враховує ті, що перебувають у процесі виконання. Ці проблеми можна легко вирішити, зберігаючи поточні саги в іншій таблиці та повністю повертаючись до попереднього стану або просто використовуючи дані, коли це необхідно.

Існує два способи реалізації Саг: оркестрація та хореографія. Не те, щоб один був кращим за інший, що є досить поширеним повідомленням, обидва використовуються в різних ситуаціях. Різниця між ними полягає в різниці між командами та подіями, це буде пояснено трохи пізніше, але загалом ми вирішуємо, хто відповідає за виклик і з скількома компонентами ми спілкуємося.

Послідовність.

Бувають випадки, коли нам потрібна точність даних і коли потрібна сильна узгодженість, або MS потрібно об'єднати в одну, або використовувати синхронний виклик. Однак, як уже було сказано, у більшості випадків ми допускаємо остаточну узгодженість даних — користувач міг бути знижений до попереднього рівня. Тим не менш, це не проблема, якщо ми дозволимо йому використовувати функцію, доки зміна плану не буде поширена.

Місце обслуговування.

Чи є учасники:

- Внутрішня — наша інфраструктура, може використовувати різноманітні технології.
- Зовнішні – зовнішні клієнти підписуються на отримання подій або

використовують інші засоби, такі як вебхуки та RSS, з розумними тайм-аутами та винятковою обережністю.

Тип запиту.

Не всі запити однакові, іноді ми просто отримуємо дані, іноді змінюємо стан, а іноді просто повідомляємо зовнішні слухачі.

Запити – запити на отримання певних даних, коли клієнт знає цільовий сервіс та схему, і отримує повернене корисне навантаження.

- Команди – команди – це запити на зміну чогось, коли клієнт знає цільову службу та схему і отримує повернене корисне навантаження.
- Події – події показують щось, що сталося, коли видавець не дбає про наслідки (що спрацьовує в інших частинах програми та результати цього). Вони не мають визначеного приймача, а також не мають корисного навантаження відповіді. Незважаючи на асинхронність, обробка подій не займає багато часу, якщо в системі немає проблеми; для виявлення таких ситуацій має бути присутня спостережуваність. Однак події є набагато складнішими, оскільки дані споживача та деталізація подій заздалегідь невідомі. Якщо ми створюємо подію для кожної окремої дії, це може просто скопіювати логіку обробки від виробника події до споживача, оскільки споживач повинен розуміти порядок обробки повідомлень та їх значення. Крім того, дані, що використовуються в подіях, можуть об'єднуватися, особливо якщо подія містить більше даних, необхідних одному споживачеві, який намагається задовольнити кілька.

Спочатку нам потрібно зрозуміти, чи мають послуги однакову причину та швидкість змін, чи можемо ми ще більше роз'єднати їх, чи, можливо, вони повинні бути одним цілим і запобігати балакучості. Те, що змінюється разом, і йде рука об руку.

З MS ми хочемо забезпечити якомога менший інтерфейс. Тому нам слід подумати про те, чим ми хочемо ділитися, і створити ATL, щоб запобігти розголошенню приватних, внутрішніх знань сервісу.

Розкриваючи події домену, ми дозволяємо споживачеві зрозуміти та обробити подію, що вводить зв'язок. Якщо лише один інший сервіс розуміє мову

(один процес зупинився тут, інший запустився там), ми можемо дозволити це, але якщо таких сервісів кілька, це може створити проблему.

Для споживача не завжди зрозуміло, що означають події. Тому важливо зробити їх більш зрозумілими, усунувши неоднозначність.

Типи подій.

Для кращого розуміння, існують різні типи подій залежно від корисного навантаження:

- Подія сповіщення (або поверхнева подія) – містить лише ідентифікатори; чим менше даних має подія, тим менше зусиль потрібно для підтримки схеми. Недоліком є те, що сервіс, який публікує подію, буде завалений запитами на додаткові дані.

Передача стану, що переноситься подією – містить корисне навантаження, яке може мати різні форми: лише оновлені поля, дельта. Цей тип події підвищує доступність, оскільки споживач отримує всі необхідні дані, що містяться в події, але також створює проблеми з обслуговуванням схеми та проблеми з узгодженістю зовнішнього кешу. Враховуючи переваги та недоліки, правильним способом є використання подій між BC, але використання ACL для зіставлення подій з більш конкретними з належною деталізацією та корисним навантаженням для різних BC.

Порядок подій.

Деякі події можуть оброблятися не в певному порядку, тоді як для інших порядок є критичним. Повідомлення про невідповідний порядок не є рідкістю, враховуючи, що присутні кілька виробників або споживачів подій, існує ймовірність того, що подія або опублікована пізніше, або використана пізніше, ніж відбулася дія (порівняно з іншими діями) [19].

Такі ситуації можна пом'якшити, використовуючи розділи та призначаючи подію певному розділу на основі даних повідомлення, таких як ідентифікатор користувача, але цього важко досягти в деяких системах, таких як Rabbit MQ, де, порівняно з Kafka, немає вбудованої підтримки розділів.

Емпіричне правило полягає в тому, щоб розробляти систему таким чином, щоб вона не залежала від порядку повідомлень. Наприклад, публікувати наступну подію лише після того, як попередня була оброблена та було надіслано підтвердження (за шаблоном Saga), або розглядати випадки, коли події відбуваються не в порядку. Наприклад, замовлення може бути скасовано до його прийняття. Зазвичай важко продумати всі можливості, тому генеративне тестування може бути корисним.

Зв'язок між подіями та командами з точки зору залежностей.

Як події, так і команди вводять залежність. Коли клієнт надсилає команду, він знає схему і має бути змінений, коли це відбувається. Події також мають схему, різниця полягає в тому, що тепер видавець нічого не знає про споживача, тому залежність змінюється на протилежну, але зв'язок залишається. В обох випадках нам потрібно відстежувати, хто споживає API, щоб знати, що змінювати. Це те саме, що й принцип інверсії залежностей у багаторівневій архітектурі, ми використовуємо його для перенесення відповідальності на правий бік.

Ідемпотентність для команд та подій.

Через те, як працюють DS, нерідко веб-сервіс отримує два однакові запити замість одного. Ця проблема може виникнути, коли ми робимо синхронний запит, але у відповіді виникає мережевий розділ, тому ми повторюємо спробу. Те саме стосується асинхронних систем: коли повідомлення публікується брокеру, а потім виникає мережевий розділ, ми повторюємо спробу публікації, або коли ми споживаємо повідомлення і не можемо підтвердити брокера (або самого брокера, якщо інше не гарантовано).

Проблему можна вирішити введенням ідемпотентності:

- Здійснюючи upsert (оновлення та вставку), не має значення, скільки разів ми виконуємо операцію.
- Збережіть ідентифікатор оброблених запитів і перевірте, чи він ще не оброблений (ключ ідемпотентності).

Airbnb реалізував «Orpheus», бібліотеку ідемпотентності загального призначення, для кількох платіжних сервісів з ключем ідемпотентності, який передається у фреймворк як єдиний ідемпотентний запит.

3.3 Аналіз ефективності сучасних підходів

У цій роботі ми проаналізували кілька дослідницьких моделей для дослідження надійності та стійкості розподілених застосунків до управління збоями в різних середовищах, таких як SOA, Grid та Cloud. Аналіз цих моделей показав, що надійність може бути забезпечена шляхом:

1. Прогнозування надійності на ранньому рівні архітектури для кращого аналізу системи перед фактичною розробкою застосунку.
2. Забезпечення відмовостійкої системи для забезпечення безперебійного середовища для кінцевих користувачів у разі виникнення будь-якого непередбачуваного або непередбачуваного сценарію під час виконання в реальному середовищі.

Ми виявили, що для розподілених застосунків на основі SOA вищезгадані методи стійкості до відмов базуються на різних стратегіях реплікації, включаючи балансувальник навантаження та реплікацію даних. Більшість цих методів просто перенаправляють запити клієнтів на резервні сервери у разі будь-якого збою. Однак для розподіленої системи реального часу, яка обробляє фінансові транзакції через Інтернет, такі як онлайн-банкінг, електронна комерція тощо, у нас все ще є деякі відкриті питання, такі як:

1. Підтримка станів служб все ще залишається проблемою, яку потрібно вирішити, оскільки іноді запити, що обробляються, не відновлюються після виникнення збою. Досі потрібно багато роботи для обробки таких оновлень, коли оновлення вже оброблялися під час виникнення збою або коли різні оновлення були випущені одночасно.
2. Ситуації, коли проблеми з мережею можуть стати причиною системного збою, оскільки навіть у сучасних мережах, що складаються з

різноманітних технологій та конфігурацій, не можна гарантувати своєчасну передачу повідомлень або відсутність випадкових збоїв [44] .

3. Іноді сервіси розміщуються іншою організацією [3] , тому необхідно забезпечити надійний рівень виявлення та відновлення помилок у разі збою цих сервісів, а також у випадках, коли неможливо вести журнал транзакцій для каскадних викликів сервісів.

У кількох моделях обговорювалося ведення журналів транзакцій на резервних серверах для відновлення систем з однієї точки відмови, забезпечуючи безперебійну взаємодію з кінцевими користувачами, проте ці методи мають свої власні накладні витрати на ведення журналів, модифікацію програм ядра Linux або модулів веб-сервера Apache та використання різних резервних серверів, що реєструють кожну окрему транзакцію між службами або між клієнтом і сервером. Однак, у випадку віддаленого веб-сервісу, існуючі методи стійкості до збоїв вирішували різні проблеми, реплікуючи один і той самий набір служб на різних географічно розділених серверах. Для забезпечення надійності в грід-системах було запропоновано різні методи стійкості до збоїв, зосереджені на різних функціональних областях, таких як апаратне забезпечення, програмне забезпечення, служби робочих процесів, мережі тощо грід-систем, проте все ще мало областей, які потребують більшої уваги для забезпечення стійкості грід-системи до збоїв. Однак занадто багато роботи було виконано в розробці метрик для грід-мереж:

1. Менше уваги приділяється розробці метрик для інших функціональних областей грід-середовища, і тому без забезпечення рішень для різних областей грід-обчислень разом важко забезпечити механізм для вимірювання надійності всього грід-середовища.

2. Різні моделі відмовостійкості в мережевих системах були запропоновані або експериментально випробувані лише для мереж малого масштабу, проте ці моделі/методи для великомасштабних гетерогенних систем все ще потребують подальшого дослідження.

3. Для вимірювання різних існуючих моделей надійності грід-

середовища було використано обмежені дані тестового середовища, проте різні області для масштабованих, гетерогенних та динамічних грід-систем, заснованих на географічно розділених серверах, потребують подальшого тестування з використанням масивів даних у реальному середовищі.

Щодо хмарних корпоративних застосунків, користувачі висловлюють серйозні занепокоєння щодо їхньої надійності, оскільки їхні дані, застосунки та обчислювальні ресурси більше не будуть під їхнім контролем. Хоча це призвело до значного покращення обробки великих обсягів даних за допомогою різних стандартних серверів, це спричиняє серйозні проблеми, коли виникають непередбачені простої в операційній системі, програмному забезпеченні, апаратному забезпеченні або мережах тощо.

1. Потрібно виконати багато роботи для забезпечення надійних хмарних сервісів, дозволяючи різні альтернативи у разі простоїв та збоїв у хмарних сервісах.

2. Щоб подолати проблеми взаємодії між різними хмарними додатками, хоча й було створено Форум взаємодії хмарних обчислень [47] , ця галузь потребує подальшого дослідження з точки зору масштабованості та гетерогенності.

3. Для забезпечення надійного масштабованого сховища в хмарному середовищі було зроблено менше роботи для диференціації різних методів масштабованості, наприклад, горизонтальної та вертикальної масштабованості, з точки зору їхніх функцій та вартості, більше роботи потрібно виконати для забезпечення надійності масштабованого сховища в хмарному середовищі з мінімальними витратами.

Прогнозування надійності розподіленого корпоративного застосунку на ранніх стадіях не лише зменшує витрати, час та зусилля на реінжиніринг, але й допомагає зробити застосунки надійними. Тому необхідно дослідити надійність усіх факторів, що впливають на надійність усіх систем, проте різні доступні методи не враховують усі фактори (як зазначено в таблиці 3) разом. Підходи/моделі, розглянуті вище з точки зору визначення надійності різних

факторів, можна приблизно класифікувати на орієнтовані на користувача, архітектурні та станоорієнтовані моделі. Оскільки надійність можна розглядати як ймовірність [40] , що означає, що її можна назвати випадковим явищем, тому дуже важко та складно виміряти або передбачити надійність будь-якого масштабованого, гетерогенного розподіленого застосунку точним та достовірним способом, а через непередбачуваний Інтернет компанії вимагають, щоб надійність цих застосунків оцінювалася та прогнозувалася постійно. Однак нескладно забезпечити адекватний показник надійності для будь-якого високонадійного великомасштабного корпоративного застосунку. Моделі, створені в цьому відношенні, намагалися зосередити увагу на певних факторах, що впливають на надійність, таких як час обробки процесора, затримка мережі, взаємодія із зовнішніми службами під час оркестрації служб тощо. Однак на ранній стадії виконання будь-якої розподіленої програми певні фактори, такі як навантаження користувача, навантаження процесора та мережевий трафік, мають значно менший вплив на загальну надійність програми, яка поступово зростає. Тому для прогнозування надійності:

1. Необхідно враховувати всі фактори залежно від впливу їх використання на надійність системи.

2. Потрібно виконати багато роботи в поєднанні з іншими факторами надійності, щоб перевірити надійність мережевих топологій; сумісність комунікаційних протоколів, таких як багаторівнева архітектура OSI, з протоколом, що використовується в розробці розподілених додатків, які можуть бути певним типом служби операційної системи.

3. Найчастіше різними розробниками під час розробки розподілених додатків у мережевому протоколі використовується комунікаційний рівень, який зазвичай приховує властивості комунікаційного обладнання, що може призвести до надсилання або отримання великих повідомлень, які зазвичай не підтримуються базовим обладнанням. Це може стати причиною збою певного модуля запущеної програми через збій доставки повідомлень, спричинений буферним простором. Хоча існують надійні канали зв'язку, які забезпечують

рівень над протоколом надійності повідомлень і можуть гарантувати доставку повідомлень, навіть якщо надійність каналу зв'язку необхідно враховувати разом з іншими факторами під час прогнозування надійності всієї системи.

3.4 Висновок по розділу

У третьому розділі розглянуто практичні аспекти реалізації та оцінювання ефективності методів забезпечення надійності розподілених систем. Проаналізовано сучасні платформні технології для створення надійних програмних рішень та досліджено особливості багаторівневої архітектури, що забезпечує ефективну взаємодію процесів і компонентів системи. Проведений аналіз показав, що використання сучасних архітектурних підходів, засобів моніторингу та механізмів відновлення після збоїв дозволяє суттєво підвищити рівень надійності та доступності програмного забезпечення. Встановлено, що впровадження комплексних методів забезпечення надійності є важливою умовою успішного функціонування сучасних розподілених програмних систем.

ВИСНОВКИ

У ході виконання бакалаврської роботи було досліджено та проаналізовано методи забезпечення надійності розподілених програмних систем, що є важливим напрямом сучасної інженерії програмного забезпечення. Робота була спрямована на вивчення підходів до побудови відмовостійких, масштабованих і ефективних систем, здатних функціонувати в умовах нестабільного середовища та високих навантажень.

У процесі дослідження було проведено аналіз предметної області, визначено основні поняття, характеристики та компоненти розподілених систем. Особливу увагу приділено принципам побудови надійних систем, зокрема забезпеченню відмовостійкості, узгодженості даних і ефективної взаємодії між компонентами. Було розглянуто базові комунікаційні служби та протоколи, які відіграють ключову роль у функціонуванні розподілених систем.

У другому розділі досліджено причини виникнення збоїв у розподілених системах, включаючи апаратні, програмні та мережеві фактори. Проаналізовано методи виявлення, діагностики та обробки відмов, а також підходи до забезпечення безперервності роботи системи. Окремо розглянуто технології обміну даними, що впливають на надійність системи, зокрема механізми синхронізації, реплікації та балансування навантаження.

У третьому розділі було розглянуто практичні аспекти реалізації надійних розподілених систем, включаючи використання сучасних платформних технологій, багаторівневих архітектур і підходів до організації взаємодії між процесами. Проведено аналіз ефективності сучасних архітектурних рішень, таких як мікросервісна та serverless-архітектури, що дозволяють підвищити гнучкість і масштабованість системи.

Тестування та оцінка ефективності досліджених підходів показали, що застосування сучасних методів забезпечення надійності дозволяє значно зменшити кількість відмов, підвищити доступність системи та забезпечити стабільність її функціонування. Використання механізмів резервування,

автоматичного відновлення, моніторингу та балансування навантаження сприяє підвищенню продуктивності та ефективності роботи розподілених систем.

Загалом, поставлену мету роботи досягнуто: досліджено основні методи забезпечення надійності розподілених програмних систем та проаналізовано їх ефективність у сучасних умовах. Перевагами розглянутих підходів є можливість забезпечення безперервної роботи системи, адаптивність до змін навантаження та гнучкість у розгортанні. Водночас існують певні обмеження, зокрема складність реалізації, підвищені вимоги до ресурсів і необхідність додаткових механізмів забезпечення безпеки.

У подальшому результати роботи можуть бути розширені шляхом впровадження інтелектуальних методів аналізу та прогнозування збоїв, використання технологій машинного навчання для оптимізації роботи систем, а також інтеграції сучасних інструментів моніторингу та автоматизації, що дозволить ще більше підвищити рівень надійності розподілених програмних систем.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Tanenbaum, A. S., & van Steen, M. (2017). *Distributed Systems: Principles and Paradigms*. Pearson. https://www.distributed-systems.net
2. Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). *Distributed Systems: Concepts and Design*. Addison-Wesley.
3. Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media. https://dataintensive.net
4. Newman, S. (2021). *Building Microservices*. O'Reilly Media.
5. Burns, B., & Beda, J. (2019). *Kubernetes: Up and Running*. O'Reilly Media.
6. Microsoft. (2025). *Azure Architecture Center*. https://learn.microsoft.com/en-us/azure/architecture/
7. AWS. (2025). *Reliability Pillar – Well-Architected Framework*. https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/
8. Google Cloud. (2025). *Site Reliability Engineering (SRE) Book*. https://sre.google/sre-book/
9. Nygard, M. (2018). *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf.
10. Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. https://martinfowler.com
11. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns*. Pearson.
12. OWASP Foundation. (2024). *OWASP Top Ten*. https://owasp.org/www-project-top-ten/
13. Microsoft. (2025). *.NET Microservices Architecture Guide*. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/
14. Red Hat. (2024). *Microservices Architecture Guide*. https://www.redhat.com/en/topics/microservices
15. IBM. (2024). *What is Distributed Computing?* https://www.ibm.com/topics/distributed-computing

16. Brewer, E. (2012). *CAP Theorem Revisited*. IEEE Computer.

17. Vogels, W. (2009). *Eventually Consistent*. Communications of the ACM.

18. Chen, L., & Ali Babar, M. (2014). *Architecting Microservices*. IEEE Software.

19. Pritchett, D. (2008). *BASE: An ACID Alternative*. ACM Queue.

20. Apache Software Foundation. (2025). *Apache Kafka Documentation*. https://kafka.apache.org/documentation/

21. Apache Software Foundation. (2025). *Apache ZooKeeper*. https://zookeeper.apache.org

22. Docker Inc. (2025). *Docker Documentation*. https://docs.docker.com

23. Kubernetes. (2025). *Official Documentation*. https://kubernetes.io/docs/

24. Netflix. (2024). *Hystrix Circuit Breaker Pattern*. https://github.com/Netflix/Hystrix

25. Resilience4j. (2025). *Fault Tolerance Library*. https://resilience4j.readme.io

26. Istio. (2025). *Service Mesh Documentation*. https://istio.io/latest/docs/

27. Linkerd. (2025). *Service Mesh for Kubernetes*. https://linkerd.io

28. Newman, S. (2020). *Monolith to Microservices*. O'Reilly Media.

29. Erl, T. (2016). *Service-Oriented Architecture: Concepts, Technology, and Design*. Pearson.

30. Kratzke, N. (2018). *Cloud-Native Microservices*. IEEE Software.

31. Zhang, Q., Chen, M., Li, L., & Li, M. (2010). *Cloud Computing: State-of-the-Art*. Journal of Internet Services and Applications.

32. Hohpe, G., & Woolf, B. (2003). *Enterprise Integration Patterns*. Addison-Wesley.

33. Microsoft. (2025). *Azure Service Bus Documentation*. https://learn.microsoft.com/en-us/azure/service-bus/

34. Amazon. (2025). *Amazon SQS Documentation*. https://docs.aws.amazon.com/sqs/

35. Google. (2025). *Cloud Pub/Sub Documentation*. https://cloud.google.com/pubsub/docs

36. Laprie, J.-C. (1992). *Dependability: Basic Concepts and Terminology*. Springer.

37. Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). *Basic Concepts and Taxonomy of Dependable Systems*. IEEE Transactions.

38. Gray, J., & Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

39. Pat Helland. (2007). *Life Beyond Distributed Transactions*. CIDR Conference.

40. Dean, J., & Barroso, L. (2013). *The Tail at Scale*. Communications of the ACM.

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврсьої роботи: "Методи забезпечення надійності розподілених програмних систем"

Обсяг пояснювальної записки: 89 аркушів

Дата закінчення бакалаврсьої роботи 10червня 2026р.

Підпис студента _____