КВАЛІФІКАЦІЙНА РОБОТА

Група ІПЗс-20
Луканюк Л.В.

2024

ЗАКЛАД ВИЩОЇ ОСВІТИ
УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук
Кафедра інформаційних технологій

на правах рукопису

Луканюк Любомир Володимирович

УДК 004.4

Імплементація моделей автономних обчислень на основі фреймворку з децентралізованою архітектурою

Спеціальність 121 – «Інженерія програмного забезпечення»

Кваліфікаційна робота на здобуття освітнього ступеню бакалавра

Науковий керівник
к.ф-м.н., доц
Бойчук А.М.

Івано-Франківськ – 2024

ЗВО “Університет Короля Данила”
Факультет суспільних та прикладних наук
Кафедра інформаційних технологій

Освітній ступінь: «бакалавр»
Спеціальність: 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____
« ____ » _____ 2024 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Луканюку Любомиру Володимировичу

(прізвище, ім’я, по батькові)

1. Тема роботи

Імплементація моделей автономних обчислень на основі фреймворку з децентралізованою архітектурою

керівник роботи:
Бойчук Андрій Михайлович, к.ф-м.н., доцент.

затверджена наказом вищого навчального закладу від « __» травня 2024 року № ___/1-НВ

2. Строк подання студентом роботи 01.06.2024

3. Зміст кваліфікаційної роботи (перелік питань, які потрібно розробити)

1. *Теоретичні основи автономних обчислень*
2. *Реалізація фреймворку автономних обчислень з децентралізованою архітектурою*
3. *Імплементація децентралізованих архітектур в автономні обчислення*
4. *Вибір методів оцінки побудованої методології*

4. Дата видачі завдання 10.02.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Теоретичні основи автономних обчислень	10.02.2024	Виконано
2.	Реалізація фреймворку автономних обчислень	24.02.2024	Виконано
3.	Імплементація моделей децентралізованих архітектур в систему автономного обчислення	10.03.2024	Виконано
4.	Вибір методів оцінки побудованої методології	24.03.2024	Виконано
5.	Формування висновків	25.04.2024	Виконано
6.	Оформлення пояснювальної записки	22.05.2024	Виконано
7.	Оформлення графічного матеріалу та підготовка до захисту роботи	12.06.2024	Виконано

Студент _____ Луканюк Л.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Бойчук А.М.
(підпис) (прізвище та ініціали)

Вихідні дані:

Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

АНОТАЦІЯ

Кваліфікаційна робота присвячена імплементація моделей автономних обчислень на основі фреймворку з децентралізованою архітектурою.

Виконано дослідження моделей, методів та алгоритмів систем з децентралізованою архітектурою і процесів автономних обчислень. Отримання глибшого розуміння та порівняння різних моделей і їх застосування при розробці фреймворку автономних обчислень. Вивчення переваг, обмежень, ефективності та відмінностей між різними підходами до побудови децентралізованих систем. Розробка висновків та рекомендацій, які можуть бути використані для вибору оптимального підходу до конкретних завдань машинного навчання та систем автономних обчислень.

В результаті досліджень було виявлено, що вибір алгоритму впливає на якість та швидкість навчання моделі, а алгоритми, які використовують градієнтний спуск, можуть бути ефективними при розробці систем автономних обчислень для фрейморків з децентралізованою архітектурою під час обробки великих обсягів даних.

КЛЮЧОВІ СЛОВА: АВТОНОМНЕ ОБЧИСЛЕННЯ, АРХІТЕКТУРА, ПЛАНУВАННЯ, ЖИТТЄВИЙ ЦИКЛ, КООРДИНАЦІЯ, КЕРОВАНА СИСТЕМА, ОЦІНЮВАННЯ, КОНТЕКСТ, АБСТРАКЦІЯ

SUMMARY

The qualification work is devoted to the implementation of autonomous computing models based on a framework with a decentralized architecture.

The study of models, methods and algorithms of systems with a decentralized architecture and autonomous computing processes was carried out. Gaining a deeper understanding and comparison of different models and their application in the development of an autonomous computing framework. Exploring the benefits, limitations, effectiveness, and differences between effective approaches to building decentralized systems. Development of conclusions and recommendations that can be used to choose the optimal approach to specific tasks of machine learning and autonomous computing systems.

In the results of the study, it was found that the choice of algorithm affects the quality and speed of model learning, and algorithms that choose gradient descent can be effective in developing an autonomous computing system for frameworks with a decentralized architecture when processing large amounts of data.

KEY WORDS: AUTONOMOUS COMPUTING, ARCHITECTURE, PLANNING, LIFE CYCLE, COORDINATION, CONTROLLED SYSTEM, EVALUATION, CONTEXT, ABSTRACTION

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ECA - політики подія-умова-дія

QoS - якість обслуговування

SML - System modeling language

EMF - Eclipse Modeling Framework

UUID - Universal Unique IDentifier

OSF - Open Software Foundation

ВСТУП

Актуальність теми. Актуальність роботи на тему "Імплементація моделей автономних обчислень на основі фреймворку з децентралізованою архітектурою" зумовлена кількома ключовими факторами, що відображають сучасні тенденції у сфері інформаційних технологій та їхнього впливу на різні галузі:

1. Зростання обсягів даних. У сучасному світі обсяги даних стрімко зростають, що вимагає ефективних методів їх обробки. Децентралізовані архітектури дозволяють розподіляти обчислювальні ресурси та зменшувати навантаження на центральні сервери, забезпечуючи швидку та масштабовану обробку великих обсягів інформації.

2. Розвиток Інтернету речей (IoT). Збільшення кількості пристроїв, підключених до Інтернету, створює необхідність в автономних обчисленнях для обробки даних на місці (edge computing), що зменшує затримки та підвищує продуктивність систем. Децентралізовані моделі обчислень ідеально підходять для таких середовищ.

3. Безпека та надійність. Децентралізовані архітектури підвищують стійкість систем до збоїв і кібератак завдяки розподілу даних і обчислювальних процесів між багатьма вузлами. Це робить системи менш уразливими до точкових відмов і забезпечує вищий рівень безпеки.

4. Ефективне використання ресурсів. Децентралізовані моделі обчислень дозволяють ефективно використовувати доступні ресурси, оптимізуючи їх розподіл і зменшуючи енергоспоживання. Це особливо важливо в контексті екологічних викликів і прагнення до створення "зелених" технологій.

5. Гнучкість і масштабованість. Децентралізовані системи забезпечують високу гнучкість і легкість масштабування, що дозволяє швидко адаптуватися до змінних потреб бізнесу та ринку. Це особливо важливо в умовах динамічного розвитку технологій і конкурентного середовища.

З огляду на зазначені фактори, дослідження та імплементація моделей автономних обчислень на основі фреймворку з децентралізованою архітектурою

є актуальними і важливими для подальшого розвитку інформатики, кібербезпеки, бізнесу та багатьох інших галузей.

Метою роботи є розкриття та розуміння основних аспектів застосування методів побудови фреймоврків з децентралізованою архітектурою для розробки алгоритмічного забезпечення системи автономних обчислень.

Досягнення мети включало розв'язання таких **задач**:

- Аналіз моделей автономних обчислень.

- Реалізація фреймворку автономних обчислень

- Імплементація моделей децентралізованих архітектур в систему автономного обчислення.

Об'єктом дослідження є моделі, методи та алгоритми децентралізованої архітектури побудови фреймворку.

Предметом дослідження є процеси імплементації моделей та методів функціонування децентралізованої архітектури в системи автономних обчислень.

Методи дослідження

Для ефективного виконання мети будуть використані такі методи: літературний аналіз вивчення наукових статей; експериментальні дослідження - застосування різних моделей та алгоритмів на конкретних даних для отримання практичних результатів; статистичний аналіз.

Результати роботи. В роботі виконано оптимізацію вибору моделей функціонування децентралізованих архітектур, а отримані результати дозволяють вибирати оптимальні моделі для конкретних завдань, що може покращити якість рішень у реальних застосуваннях автономних обчислень. Вибір оптимальних алгоритмів - результати дослідження допомагають визначити оптимальні алгоритми для різних задач, що сприяє підвищенню ефективності обчислень. Отримання глибшого розуміння та порівняння різних моделей автономних обчислень, їх методів та використання алгоритмів для побудови фрейморку з децентралізованою архітектурою.

Структура роботи. Розділи – 3. Загальний обсяг основної частини – 81 сторінка. Список використаних джерел – 41.

РОЗДІЛ 1. Аналіз предметної області здійснення автономних обчислень

1.1 Важливість і проблеми управління програмним забезпеченням

Системи програмного забезпечення стали центральною частиною асортименту продуктів і послуг, що швидко зростає, у всіх секторах. Сьогодні такі системи необхідні в багатьох сферах нашого життя. Ми використовуємо їх удома, для бізнесу, у лікарнях, для банківської справи тощо. На жаль, розробка та керування такими програмними системами стає дедалі складнішим і дорогим.

З інженерної точки зору, першою проблемою при розробці програмних систем раніше було пошук найкращого алгоритмічного рішення для виконання певних операцій. Згодом програмне забезпечення було розподілено між кількома фізичними машинами для покращення нефункціональних якостей, таких як продуктивність, масштабованість або доступність. Це призвело до іншого рівня складності ; фокус змістився зі складності розробки алгоритмів на складність побудови розподілених паралельних систем і структурування великомасштабних систем, таких як хмара [9] і мережеві обчислення [3]. У наш час програмні системи взаємодіють з іншими системами, пристроями та людьми – тобто перетворюються на соціально-технічні системи – розподілені у великих зонах розгортання, можливо, по всьому світу.

Отже, системи програмного забезпечення стають все більш розподіленими, гетерогенними, децентралізованими та взаємозалежними , і вони все більше і більше працюють у динамічних і часто непередбачуваних середовищах. Ця ситуація вимагає від системних архітекторів і розробників зробити системи програмного забезпечення більш адаптивними та контекстно-орієнтованими (рис. 1.1), з недоліком збільшення складності розробки та управління. З точки зору системного управління та контролю, що є центром роботи, програмні

системи, таким чином, стають все більш складними, а тому їх розробка та підтримка складніша та дорожча.

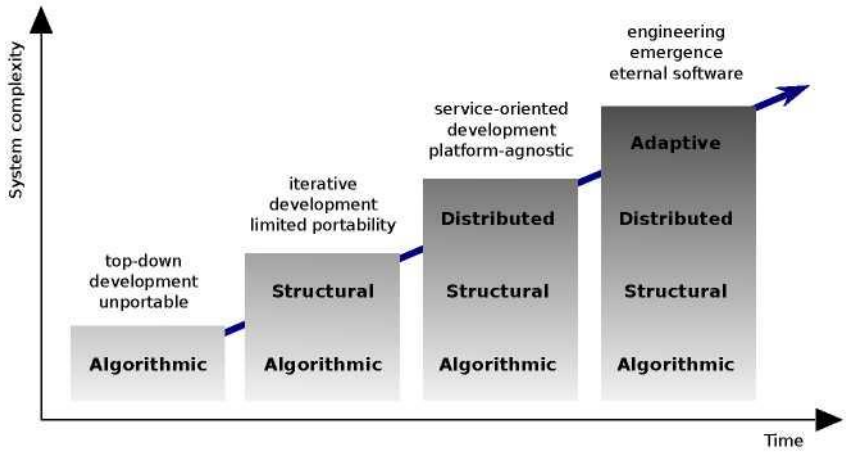


Рисунок 1.1 - Зростання складності програмного забезпечення

З часом (управління та контроль ми маємо на увазі операції адміністрування, які охоплюють всю обчислювальну екосистему, включаючи комп'ютерне обладнання, мережу та програмне забезпечення.

Системне адміністрування стає складнішим з кількох причин. По-перше, програми часто неоднорідні, складаються з кількох інтегрованих підсистем, таких як бази даних і сервери програм. Експериментований оператор потрібен для кожної підсистеми, щоб підтримувати її правильне й оптимальне виконання. Фахівці з кожної технології повинні спілкуватися та ділитися знаннями для досягнення цілей глобального управління. По-друге, багато додатків розповсюджені та побудовані на складному проміжному програмному забезпеченні, яке потрібно налаштовувати та переналаштовувати під час виконання. Наприклад, використання EJB [6] може значно спростити роботу розробників, але ціною вищих витрат на адміністрування. Дійсно, складні - декларативні дескриптори розгортання повинні бути створені під час розгортання та оновлені під час виконання. Необхідність мати справу з гетерогенним проміжним програмним забезпеченням, запроваджуючи проміжне програмне забезпечення для технологій проміжного програмного забезпечення,

таких як веб-сервіси, ще більше ускладнює ситуацію. По-третє, збільшення масштабів таких систем, як за кількістю компонентів, так і за територією розгортання, у багато разів посилює вищезгадані проблеми. І останнє, але не менш важливе: системи, що відносяться до нових областей, таких як повсюдні та повсюдні обчислення, працюють у динамічних, неоднорідних і часто непередбачуваних середовищах. У таких випадках контекст виконання системи під час розгортання, ймовірно, різко зміниться протягом усього життя програми. Як наслідок, під час виконання потрібні коригувальні операції адміністрування .

Виходячи з цих міркувань, ми виділяємо ключові виклики управління системою, які необхідно вирішити в наступному.

Неоднорідність ресурсів. Адміністрації доводиться мати справу з різнорідними даними, елементами обробки даних і підтримкою зв’язку від різних постачальників, які базуються на різних мовах і виконуються на різних апаратних і програмних платформах. Багато частин вважаються застарілими системами, тоді як інші є більш відкритими та гнучкими. Для ефективного адміністрування таких систем необхідні глибокі знання гетерогенних технологій.

Динамічність. Багато обчислювальних систем працюють у високодинамічних середовищах, тому зазнають частих змін як у своїх внутрішніх компонентах, так і в умовах їх виконання. Наприклад, у всеосяжних обчисленнях пристрої можуть приєднатися до системи або залишити її в будь-який час. Деякі пристрої можуть бути невідомі системі до їх інтеграції під час виконання. У всіх випадках глобальне програмне забезпечення система повинна продовжувати своє виконання та справлятися з такими динамічними явищами без значних перерв.

Збільшення кількості інтегрованих підсистем. Сучасні системи програмного забезпечення дозволяють інтегрувати кілька пристроїв, таких як смартфони або домашні пристрої, які включають різноманітні датчики та виконавчі механізми. Відбувся поступовий перехід від погляду на централізовану систему до сильно децентралізованого комп’ютерного ландшафту. А саме, у новіших підходах обчислення розподіляються між різними

відносно невеликими пристроями, кожен з яких відповідає за певні завдання. Збільшення кількості та розповсюдження таких пристроїв у поєднанні з проблемами гетерогенності та динамічності, зазначеними вище, у кращому випадку ускладнюють адміністрування таких програм. Відповідно , системи управління повинні мати можливість справлятися зі зростаючою складністю системи з точки зору масштабу, неоднорідності, розподілу та динамічності.

Підвищення бізнес-вимог, які часто змінюються під час виконання. Для більшості довготривалих програм вимоги користувача змінюються протягом життя системи. Наприклад, нефункціональні властивості , визначені спочатку, можуть стати невідповідними та вимагати коригування для задоволення нових потреб бізнесу. Зміни у вимогах користувача можуть вплинути на всю систему програмного забезпечення, і можуть бути надзвичайно складними для правильної та ефективної обробки.

Cube [9, 11,12] – це загальний підхід для проектування автономних систем управління, які можуть вирішити вищезазначені проблеми, наприклад неоднорідність, динамічну адаптивність, масштабованість та еволюцію бізнес-цілей. Він пропонує рішення, яке поєднує два добре відомі підходи: повністю децентралізований підхід «знизу вгору» та повністю детермінований підхід «зверху вниз». Підходи «знизу вгору» сприяють децентралізованій логіці керування, заснованій на повністю розподілених і роз’єднаних контролерах . Його основні переваги включають загальну живучість і масштабованість з точки зору потужності обробки даних. Тим не менш, складність прогнозування загальної поведінки децентралізованої системи разом із можливими явищами, що виникають , робить проектування та підтримку таких систем досить складним і дорогим. Крім того, додатковий зв’язок, необхідний для координації різних частин системи, може мати негативний вплив на масштабованість системи. І навпаки, підходи «зверху вниз» базуються на централізованій, високодетермінованій логіці керування. Тут процес прийняття рішень спирається на представлення глобальної системи та може планувати узгоджені дії щодо

компонентів керованого програмного забезпечення, отже, легше забезпечуючи правильність та ефективність операцій адаптації та самоуправління.

Підхід Cube спрямований на поєднання двох вищезгаданих підходів, щоб отримати вигоду з їх переваг, уникаючи відповідних недоліків. Зокрема, Cube покладається на децентралізовані модулі прийняття рішень, координовані за допомогою чітко визначених протоколів. Це дозволяє системі швидше приймати локальні рішення на основі поглядів локальної системи. Це дозволяє уникнути необхідності підтримувати представлення глобальної системи та повністю враховувати їх під час прийняття кожного управлінського рішення. Таким чином, споживання ресурсів розподіляється між децентралізованими процесами, що важливо при управлінні великомасштабними системами, які потребують частих адаптацій. У той же час, глобальна самоорганізаційна поведінка автономної системи Cube контролюється через формальну специфікацію мети, яка відтворюється в усіх децентралізованих процесах. Це гарантує, що загальна обчислювальна система, керована за допомогою децентралізованого підходу, завжди відповідатиме обмеженням у специфікації мети. Цей рівень детермінізму є типовою перевагою централізованого підходу.

Точніше, підхід Cube сприяє двом основним методам:

- Зверху вниз: використання абстрактної архітектурної моделі (або архетипу) для формального визначення адміністративних цілей; це має на меті контролювати процес автономного управління та гарантувати життєздатність отриманих конфігурацій системи;
- Знизу вгору: децентралізація процесу автономного управління , щоб уникнути єдиної точки контролю та пов’язаних з цим обмежень надійності та масштабованості; децентралізовані менеджери координують роботу для отримання узгоджених систем, які відповідають архітектурній моделі.

Незважаючи на те, що цей підхід виглядає багатообіцяючим, він також створює складні проблеми, які необхідно вирішити при впровадженні конкретних автономних систем. До найважливіших питань належать:

- Як спроектувати та оформити архітектурну модель? Модель має бути достатньо обмежувальною, щоб гарантувати основні властивості керованої системи, а також допускати достатню варіативність для адаптації;
- Як розподілити адміністративні завдання між децентралізованими автономними менеджерами? Динамічні зміни в керованій системі вимагають, щоб призначення завдань автономним менеджерам було адаптованим під час виконання;
- Як скоординувати діяльність автономних менеджерів, щоб забезпечити узгодженість і відповідність результуючої системи змодельованим цілям? Автономним адміністраторам, можливо, доведеться координувати роботу як локально, так і віддалено, щоб забезпечити локальні або більш глобальні властивості системи відповідно . - Як забезпечити розширюваність автономних менеджерів для підтримки майбутніх бізнес-цілей і різних типів керованих ресурсів?

1.2 Опис концепції автономного обчислення

Підхід до автономних обчислень сприяє глобальному системному підходу, який дозволяє інтегрувати та координувати численні обчислювальні системи та їхнє самокерування в цілому [1, 3]. Як ми побачимо більш детально в цій роботі, автономні обчислення охоплюють широкий спектр інструментів і методів, включаючи платформи виконання, моделі програмування та спеціалізовані алгоритми, щоб дозволити самокерування складних обчислювальних систем. Важливим завданням, звичайно, є інтеграція цих методів, щоб забезпечити відтворювані рішення.

Цей цілісний підхід виглядає на відміну від багатьох існуючих обчислювальних систем, які спеціалізуються на конкретній функції управління, такій як оптимізація кількох атрибутів або функцій (наприклад, оптимізатор запитів до бази даних). У автономному випадку завдання є більш важливим, оскільки воно прагне відобразити цілі системи як самокеровані. Наприклад,

автономний засіб оновлення для систем бухгалтерського обліку [3] має розгортати кілька автономних елементів для керування різними етапами процесу оновлення. Крім того, система Grid Computing [3] охоплює кілька гетерогенних розподілених комп'ютерів. Тут автономні елементи мають бути розгорнуті на кожному вузлі, щоб контролювати стан і контекст цього вузла, аналізувати його вимоги (відмовостійкість, продуктивність, QoS, безпека тощо) і адаптувати його для задоволення адміністративних цілей].

З іншої точки зору, автономні обчислення розглядаються як спроба об'єднати вибірку існуючих галузей з метою побудови самокерованих систем. Автономне обчислення не вказує, які типи технологій слід використовувати для досягнення цілей, але будь-яка існуюча технологія, яка демонструє поведінку або часткову поведінку поширеності та самокерування, може бути класифікована як автономне обчислення [5]. Визначають автономні обчислення як « концепцію, яка об'єднує багато областей обчислення з метою створення обчислювальних систем, які самостійно керують» [8]. Відповідно, «що є новим, так це цілісна мета Autonomic Computing — об'єднати всі відповідні сфери разом, щоб змінити напрямок розвитку галузі: автономне програмне забезпечення замість циклу оновлення функцій апаратного та програмного забезпечення минулого, який створював складність і загальну вартість болото власності" [5].

Автономні обчислення черпають натхнення з двох основних областей високого рівня: дослідження складних природних систем і розробки надскладних обчислювальних систем. Відповідні обчислювальні системи були розроблені в дуже різноманітних областях, починаючи від традиційних систем автоматизації і закінчуючи системами на основі методів штучного інтелекту. Вони надають алгоритми, архітектури, моделі та методи, які можна безпосередньо повторно використовувати в автономних обчисленнях. У свою чергу, природні системи не є програмним забезпеченням, їх вивчення може охоплювати такі різноманітні сфери, як економіка, біологія, хімія чи фізика, але вони також можуть зробити внесок в автономні обчислення з точки зору теорій і моделей.

Зокрема, розробники автономних систем можуть використовувати результати з таких областей:

- Автоматичне керування. Цей домен включає додатки теорії управління та техніки управління, які застосовувалися до механічних, електричних, хімічних або фінансових систем.
- Робототехніка. Роботи часто працюють у дуже динамічному середовищі, де здатність до адаптації є важливою, а можливості для втручання людини обмежені або небажані.
- Мультиагентні системи. Ключова перевага цих систем полягає в їхній здатності вирішувати складні обчислювальні проблеми, розділяючи та розподіляючи їх серед набору спеціалізованих агентів міркування.
- Розробка програмного забезпечення. Було впроваджено багато парадигм і технологій для забезпечення надійності та гнучкості додатків у контекстах виконання, чутливих до частих (динамічних) змін.

Взаємозв'язки між такими поняттями, як самоадаптивні, самоорганізовані та автономні системи, часто можуть викликати певну плутанину. Між цими поняттями є схожість і відмінності. Самоадаптація дозволяє системі програмного забезпечення динамічно змінювати свою структуру та поведінку у відповідь на зміни в середовищі виконання. Роботи з самоадаптації часто зосереджуються на прикладному та проміжному рівнях, тоді як автономні обчислення також охоплюють нижчі рівні, такі як операційні системи та мережа. Самоорганізація дозволяє системам, що складаються з великої кількості підсистем, виконувати колективне завдання. Сьогодні самоорганізація здебільшого застосовується до протоколів і послуг низького рівня зв’язку, тоді як ініціатива автономних обчислень охоплює великомасштабні системи, а також орієнтовані системи. Однак поняття цих областей тісно пов’язані між собою і в багатьох випадках можуть використовуватися як взаємозамінні.

У будь-якому випадку, видатною особливістю автономних обчислень є те, що вони використовують цілісний підхід до проектування та розробки обчислювальних систем, які можуть адаптуватися до мінливих умов. У нашій

дисертації ми використовуємо термін автономне обчислення, щоб охопити всі інші пов’язані області. Ми лише посилаємося на кожну конкретну область, коли детально розповідаємо про основні методи, які використовуються для демонстрації конкретних можливостей автономного обчислення.

1.3 Характеристика і завдання автономних обчислювальних систем

Існують загальні елементи або характеристики, якими повинні володіти програмні системи, щоб вважатися автономними [1]:

1. Щоб бути автономною, обчислювальна система повинна «знати себе» — і містити компоненти, які також володіють ідентифікацією системи. Тобто, щоб керувати собою, автономні системи повинні мати докладні знання про свій поточний внутрішній статус і взаємозв’язки з іншими системами та ресурсами.

2. Автономна обчислювальна система повинна налаштовуватися та переконфігуруватися за мінливих і непередбачуваних умов. Оскільки автономна система може мати сотні можливих конфігурацій, вона повинна вибрати найкращу для контексту та зробити це протягом розумного періоду часу.

3. Автономна обчислювальна система ніколи не погоджується на статус-кво — вона завжди шукає шляхи оптимізації своєї роботи. Він повинен контролювати свої показники, щоб самостійно оптимізувати свої обчислення та досягати мінливих цілей користувача за допомогою адаптивних алгоритмів, а також передових систем контролю зі зворотним зв’язком.

4. Автономна обчислювальна система повинна виконувати щось схоже на лікування — вона повинна мати можливість відновлюватися після рутинних і надзвичайних подій, які можуть спричинити збій у роботі деяких її частин. Коли виникають помилки, автономна система повинна мати можливість виявляти проблеми, а потім знаходити рішення, щоб забезпечити безперебійне функціонування. Спочатку такі реакції на самовідновлення відповідатимуть правилам, наданим експертами-людьми, але з часом вони будуть доповнені процесами самонавчання, властивими вегетативній системі.

5. Віртуальний світ не менш небезпечний, ніж фізичний, тому автономна обчислювальна система повинна бути фахівцем із самозахисту. Він повинен мати можливість виявляти та обробляти потоки від вірусів і вторгнень хакерів, щоб підтримувати загальну безпеку та цілісність системи.

6. Автономна обчислювальна система знає своє середовище та контекст, що оточує її активність

7. Автономна обчислювальна система не може існувати в герметичному середовищі. Потрібні відкриті стандарти для ідентифікації систем, зв'язку та узгодження, оскільки автономні системи працюють у гетерогенних обчислювальних середовищах, що швидко розвиваються.

8. Можливо, найважливіше для користувача те, що автономна обчислювальна система передбачає оптимізовані необхідні ресурси, зберігаючи свою складність прихованою. Це кінцева мета для автономних обчислень. Він повинен виконувати свої різні функції, зберігаючи свою складність прихованою від користувачів.

Ці вісім ключових характеристик автономних обчислювальних систем були додатково синтезовані з точки зору цілей користувача в чотири основні властивості. Кожна властивість описує мету користувача або «вимогу до якості автономної системи», яку необхідно враховувати під час розробки автономної системи:

- Самоналаштування. Автономна обчислювальна система налаштовується відповідно до цілей високого рівня, що представляють цілі бізнес-рівня. Тобто він автоматично адаптується до середовища, що динамічно змінюється, додаючи /видаляючи комп'ютери, встановлюючи/видаляючи програмні модулі, налаштовуючи нові конфігурації тощо. Наприклад, коли новий програмний модуль включається в бізнес-додаток, він автоматично інтегрується з рештою системи, включаючи необхідні конфігурації та підключення до баз даних, веб-додатків та інших екземплярів того самого модуля. Відповідно, система самостійно налаштовується плавно та без переривання обслуговування.

- Самооптимізація. Автономна обчислювальна система оптимізує використання ресурсів і працює оптимально. Він буде постійно контролювати та автоматично налаштовуватися для покращення своєї роботи. Такі види можливостей слід поширювати на різні неоднорідні складові системи. Як наслідок , налаштування підсистеми може мати непередбачені наслідки для всієї системи.
- Самовідновлення Автономна обчислювальна система виявляє, аналізує, діагностує та виправляє проблеми, що виникають через збої в апаратному забезпеченні (наприклад, проблема з жорстким диском) або програмному забезпеченні (наприклад, помилки в програмному компоненті). Він повинен бути в змозі виявити, виправити або ізолювати несправний елемент і спробувати знайти вирішення проблеми без будь-яких видимих збоїв у системі. Важливо, щоб процес самовідновлення не створював нових помилок або втрати життєво важливих налаштувань і функцій системи.
- Самозахист . Автономна обчислювальна система захищає себе від зловмисних атак, неправильного використання користувачами та ненавмисних змін програмного забезпечення, які можуть спричинити каскадні збої. Він автоматично налаштовується на захист системи в цілому від проблем безпеки, що виникають. Він також може передбачати порушення безпеки, демонструючи проактивний аналіз.
- Самодіагностика: здатність системи самоаналізувати, щоб виявити наявні проблеми або протидіяти передбачити потенційні проблеми.
- Самоадаптація: здатність системи модифікувати себе (самонастроюватися) у відповідь на зміни в її контексті виконання або зовнішньому середовищі, щоб продовжувати досягати своїх бізнес-цілей, незважаючи на такі зміни.
- Самознищення: вбудована здатність системи знищувати себе через те, що вона визначає, що більше не здатна досягати своїх цілей (наприклад, пошкоджена система вимикається, щоб запобігти впливу на безпеку користувача

або зараженню сусідніх систем); або через те, що він досяг попередньо визначеного терміну придатності.

• Самостабілізація: здатність системи досягати стабільного законного стану, починаючи з довільного стану та після кінцевої кількості кроків виконання. Ця властивість традиційно пов’язана з відмовостійкістю в розподілених системах [9], але приділяє все більше уваги спільноті систем самокерування (наприклад, забезпечення того, що операції самовідновлення або самооптимізації наближаються до стану системи, який відповідає політики високого рівня).

1.4 Еталонна архітектура автономних обчислень

Щоб реалізувати бачення автономних обчислень, дослідники IBM створили архітектурну структуру для автономних систем [3] і супровідний план [3]. Цей проект організовує автономну обчислювальну систему в будівельні блоки, які можна об’єднати, щоб надати само* властивостей. Ці будівельні блоки називаються автономними елементами.

Як пояснюється в [13], вегетативна система складається з ряду вегетативних елементів, які взаємодіють чи ні. Автономний елемент — це виконувана програмна одиниця, яка демонструє автономні властивості. Для цього він реалізує цикл керування, щоб постійно досягати цілей високого рівня, встановлених уповноваженими організаціями.

Зокрема, автономний елемент регулярно відчуває можливі джерела змін через датчики, міркує про поточну ситуацію та виконує адаптацію через виконавчі механізми, коли і де це необхідно . Будь-яка дія, яку виконує автономний елемент, робиться для того, щоб краще задовольнити свої цілі з урахуванням поточної ситуації. На цьому рівні абстракції нас цікавить лише поведінка вегетативного елемента, яке можна спостерігати ззовні, ігноруючи його внутрішню реалізацію. Дійсно, вегетативний елемент може бути раціональним (або «розумним») або цілком рефлекторним; а також жорстко

закодовані або оснащені можливостями самонавчання. Те, що вегетативний елемент робить у певний час, залежить від його цілей, від того, що він сприймає, і, можливо, від його знань.

Автономний елемент можна розглядати як самостійний програмний модуль. Однак він керується інформацією, наданою адміністраторами, людьми чи ні, включаючи цілі, які мають бути досягнуті, політику та стратегії, які мають застосовуватися. Автономний елемент повинен звітувати перед адміністраторами. Він повинен забезпечити їм стійкий зворотний зв'язок, щоб вони могли знати про його внутрішню ситуацію та ступінь успіху в досягненні своїх цілей. Таким чином, адміністратори можуть коригувати цілі або переглядати стратегії, коли це необхідно.

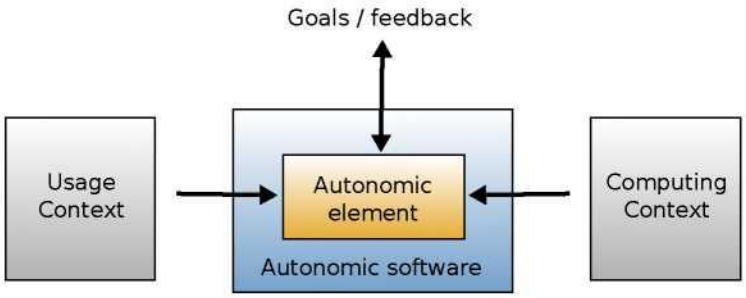


Рисунок 1.2 - Автономний елемент [13]

Автономний елемент, як правило, покладається на контур керування, який відстежує поточний стан елемента та змінює його, якщо необхідно. Зокрема, архітектура, запропонована IBM, чітко розрізняє керовані ресурси та модуль керування, який називається «автономним менеджером» (рис. 1.1). Керовані ресурси – це програмні або апаратні об’єкти, які автоматично адмініструються в автономному елементі. Вони можуть представляти, наприклад , веб-сервер, базу даних, віртуальну машину або ресурс процесора сервера. Автономний менеджер — це програмний модуль, який реалізує розширений контур керування, представлений тут — контур MAPE-K. Автономний менеджер відповідає за

адміністрування під час виконання свого керованого ресурсу та, загалом, за автономний елемент, до якого він належить.

Керовані ресурси забезпечують спеціальні інтерфейси, які називаються контрольними точками або точками дотику, для моніторингу та адаптації. Було визначено два типи точок дотику: сенсори та ефектори. Датчики надають інформацію про керовані ресурси. Вони матеріалізуються деяким кодом, який вимірює фізичну або абстрактну величину, що стосується керованого ресурсу, і перетворює її на сигнал для автономного менеджера. Це може бути деяка інформація про стан елементів або деяке уявлення про їхню поточну продуктивність. Наприклад, для веб-сервера такий показник може включати час відповіді на запити клієнта, рівень використання мережі та диска або використання процесора та пам’яті. Інші приклади таких даних включають характеристики продуктивності системи, контекст користувача або навіть температуру сервера.

Залежно від цільових вегетативних властивостей, різні типи даних і різні форми презентацій -можуть знадобитися для виконання дій із самоконтролю. Визначення відповідних даних, які потрібно зібрати, і впровадження відповідних датчиків сьогодні вважається важкою діяльністю. Він вимагає знайти відповідний баланс між кількістю зібраних даних і вартістю отримання даних (моніторинг завжди має витрати). Тому, враховуючи складність і вартість інструментування системи, метою є не збір будь-якої інформації, яку можна отримати про систему, а скоріше отримання відповідних даних, які можна використовувати для здійснення автономного керування.

Ефектори надають інтерфейси для контролю/модифікації керованих ресурсів і, як наслідок, для зміни їхньої поведінки. Наприклад, це може являти собою певну модифікацію файлу конфігурації, створення нових об’єктів , видалення компонента, заміну деяких застарілих елементів тощо. Ефектори також матеріалізуються деяким кодом, який впливає на зміни та який надається керованими елементами. Мета ефекторів полягає в тому, щоб дозволити автономному менеджеру ініціювати модифікації керованих елементів

узгодженим і контрольованим способом. Якщо ми повернемося до прикладу з веб-сервером, датчики отримуватимуть дані про завантаження сервера та кількість активних з'єднань, тоді як ефектори змінюватимуть властивості конфігурації сервера.

Метою автономного менеджера є адаптація набору керованих ресурсів під час виконання, у відповідь на внутрішні або зовнішні зміни, для досягнення заздалегідь визначених цілей. Автономний менеджер побудований навколо циклу контролю «збирати/вирішувати/діяти», який також може спиратися на певні спільні знання.

Відповідно до бачення IBM, цей еталонний цикл керування складається з набору завдань, які виконуються повторюваним способом. Цей цикл відомий під акронімом MAPE-K, що означає моніторинг, аналіз, план і виконання. K означає знання, що стосується знань, необхідних для виконання вищезазначених завдань. Цей цикл зображено на рис. 1.2. Цей підхід спирається на добре відому схему спостереження/діагностики/рішення.

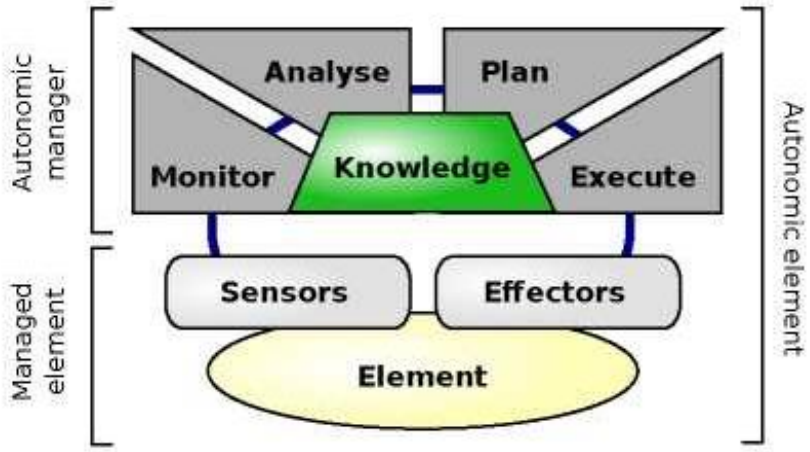


Рисунок 1.3 - Контур MAPE-K автономного обчислювального елемента

Логічна архітектура MAPE-K суттєво вплинула на сферу автономних обчислень, забезпечивши структурну структуру, з якої можна почати створення автономної системи. Це модульна архітектура, яка має сенс для практиків і поєднує в собі такі властивості, як поділ справ і масштабованість [13]. Різні види

управлінської діяльності, визначені досить абстрактно, піклуються про цілеспрямовані, чітко визначені та взаємодоповнюючі аспекти. Стандартизація комунікаційних інтерфейсів цих видів діяльності, за якою виступає IBM, дозволить легше інтегрувати різні методи, розроблені різними постачальниками. Архітектура також має певний ступінь масштабованості, оскільки дії керування можуть виконуватися на різних машинах, припускаючи, що це пов'язано із затримкою мережі та не впливає на реакцію.

Для кожного етапу циклу MAPE-K дослідники досліджували використання різних технік та інструментів. На рис. 1.3 показано деякі пов'язані роботи. Далі ми детально розглянемо ці чотири етапи.

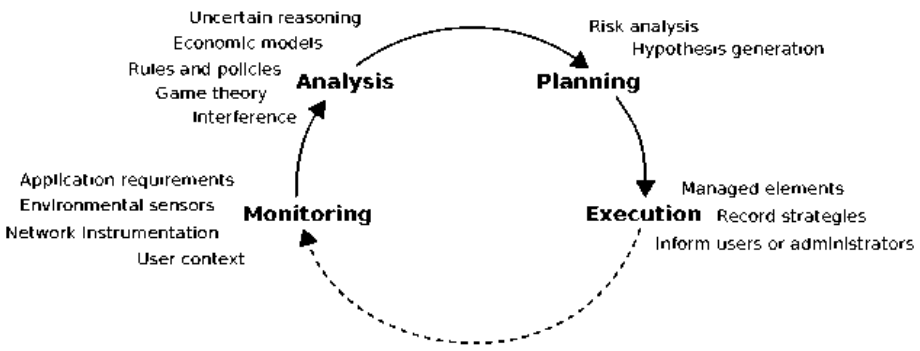


Рисунок 1.4 - Технології, застосовані до чотирьох етапів автономного менеджера [10]

Знову ж таки, рис. 1.3 підкреслює той факт, що автономне обчислення має ідентифікувати та адаптувати техніки та технології з багатьох областей. Дійсно, цей цикл «Моніторинг, аналіз, планування, виконання» явно черпає натхнення з таких областей, як теорія керування, штучний інтелект і робототехніка. Ці області фактично мають спільні точки і часто використовуються разом, наприклад, коли мова йде про проектування та впровадження автоматів і автономних роботів.

Важливо відзначити, що цикл MAPE-K представляє логічну архітектуру. Скоріше його мета полягає в тому, щоб вказати основні функції, які автономний

менеджер повинен підтримувати для адміністрування системи, і основні взаємозалежності між цими функціями, тобто аналіз залежно від моніторингу, планування від аналізу, виконання від планування, і все це залежить від спільних знань. Він також показує спосіб, у який автономний менеджер взаємодіє з керованими ресурсами - через сенсорні та ефекторні точки дотику. Для створення цієї еталонної архітектури можливі різні конкретні проекти та реалізації.

Дійсно, пропозиція MAPE-K не завжди безпосередньо застосовна. Наприклад, наявність потоку керування, що проходить через чітко визначені інтерфейси чотирьох дій управління, має витрати на продуктивність, які не завжди можна собі дозволити. Таким чином, у деяких випадках може знадобитися згрупування деяких управлінських дій, наприклад, аналізу та планування, щоб дотриматися встановленого терміну. І навпаки, реалізовувати кожну дію за допомогою одного програмного модуля не завжди можливо. Наприклад, наявність одного автономного менеджера, відповідального за кілька різнорідних ресурсів, може вимагати кількох модулів моніторингу та/або виконання.

Моніторингова частина автономного менеджера збирає, агрегує та фільтрує дані, отримані від керованого елемента та, можливо, від інших відповідних об’єктів. Завдання моніторингу безпосередньо використовують точки дотику сенсорів, доступні керованими елементами та/або середовищем виконання. Метою моніторингу є збір «корисних» даних, які забезпечують узагальнене уявлення про процес виконання та/або контекст. Такі дані, після фільтрації та відповідного форматування, потім аналізуються автономним менеджером, який потім може вжити коригувальних дій у відповідь на зміни або відхилення від цілей, встановлених адміністраторами.

Етап моніторингу включає кілька методів моніторингу, які залежать від типу інформації, яка необхідна вегетативному елементу для належного виконання своєї роботи. Наприклад, у разі розподілу ресурсів у середовищі Grid для цілей самооптимізації або самовідновлення необхідно контролювати

використання ЦП і пам’яті кожного активного вузла в Grid. Потім автономний менеджер аналізує ці дані, щоб виявити збої або неоптимальну продуктивність і, як наслідок, прийняти відповідні рішення. Датчики часто залежать від застосування. Таким чином, автономні інфраструктури, як правило, надають API спеціального призначення для реалізації конкретних датчиків для конкретних цільових систем.

Добре відома проблема полягає в тому, що моніторинг може бути досить дорогим процесом. Насправді існує компроміс між даними, які необхідні для розуміння стану системи та виконання відповідних дій, і фактичною вартістю отримання даних. Багато роботи в дослідницькому співтоваристві автономних обчислень зосереджено саме на тому, як вирішити, яку підмножину багатьох показників продуктивності, які можна зібрати з динамічного середовища, слід фактично отримати за допомогою доступних інструментів продуктивності.

Крім того, дані, які збираються, залежать від цілей і стану процесу управління. Цілі, встановлені адміністраторами, однозначно впливають на те, як слід здійснювати моніторинг. Фокус моніторингу також може змінюватися залежно від інтересів адміністраторів. Подібним чином проміжні результати щодо ситуації керованих артефактів щодо цілей можуть впливати на дані, що підлягають моніторингу, і спосіб їх збору.

Існує два види моніторингу в автономних обчислювальних системах [5]: пасивний і активний моніторинг.

Пасивний моніторинг полягає у виявленні компонента програмного або апаратного забезпечення спеціалізованим об’єктом моніторингу. Найчастіше цільові системи мають вбудовані компоненти моніторингу. У Linux, наприклад, команда 'top' повертає інформацію про використання ЦП кожним процесом. Іншим прикладом у контексті віртуальних машин, кластерів і мереж є система моніторингу [4]. Ця система покладається на протокол прослуховування/ оголошення на основі багатоадресної передачі для моніторингу стану цільових систем. Щоб проілюструвати це, на рис. 1.4 представлено поточний звіт Ganglia під час моніторингу інфраструктури Вікіпедії. Ця інформація може бути

використана автономними менеджерами для самостійного керування відповідною системою.

Метою активного моніторингу є захоплення інформації низького рівня з компонентів цільової системи за допомогою таких методів, як впровадження коду або аспектно-орієнтоване програмування. Це особливо цікавий варіант, який дозволяє автономним менеджерам вводити зонди в цільову систему, коли це необхідно, і для отримання конкретної інформації, яка їм потрібна. Цікавий приклад такого типу моніторингу можна знайти в [11]. У цій роботі набір зондів впроваджується на різних рівнях ланцюга посередництва даних для отримання відповідних даних моніторингу за потреби.

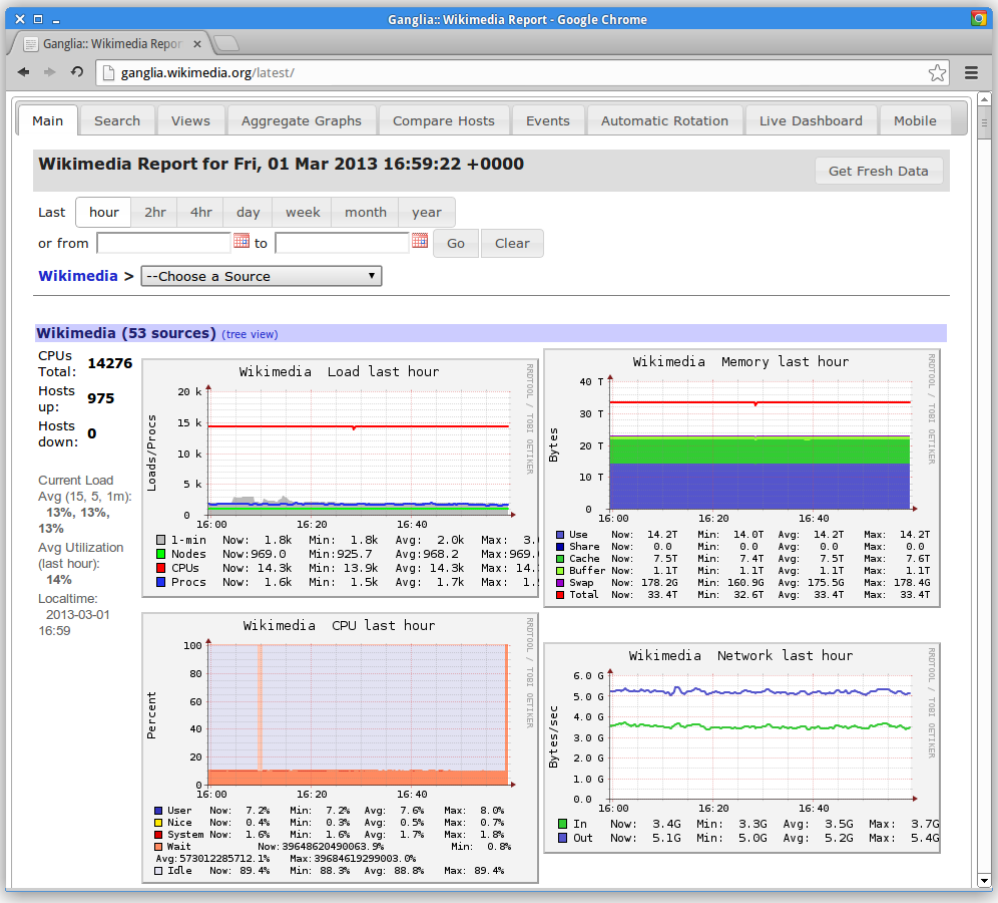


Рисунок 1.5 - Звіт системи моніторингу серверів Вікіпедії

Метою процесу аналізу є оцінка поточного стану цільової системи та середовища виконання. Для цього він використовує дані моніторингу, моделює

ситуацію та аналізує її визначити, чи потрібно вносити зміни в керовану систему. Для виявлення неправильної поведінки або недостатньої якості обслуговування, кореляції інформації, передбачення поганих ситуацій, виявлення проблем і визначення кращого стану, якого потрібно досягти, можна використовувати широкий спектр алгоритмів і методів . Таким чином, фаза аналізу може визначити бажані стани, до якого має бути зміщена керована система. Фаза планування - наступна в циклі - відповідає за підготовку набору дій, які необхідно виконати для досягнення таких станів.

Як було сказано, доступних методів для виконання аналізу безліч. Вони включають побудову моделей для оцінки ситуації, класифікаційні підходи для визначення того, коли обмеження не виконуються, або системи навчання для кращого сприйняття ситуацій заздалегідь. Системи прогнозування зазвичай відстежують тенденції та передбачають, чи буде порушено обмеження чи ціль у найближчому чи віддаленому майбутньому. Це можна реалізувати за допомогою простого регресійного аналізу або більш складних методів, таких як приховані марковські моделі, які представляють тимчасові стани системи та можуть використовуватися для моделювання результатів плану.

Висновки до розділу 1

В даному розділі виконано аналіз предметної області здійснення автономних обчислень. Представлено важливість і проблеми управління програмним забезпеченням, описано концепції автономного обчислення. Наведено характеристика і завдання автономних обчислювальних систем.

РОЗДІЛ 2. Алгоритмічна та структурна реалізація децентралізованого фреймворку автономних обчислень

2.1 Опис проектного рішення

Наша пропозиція полягає в автономній структурі управління для високодинамічних, розподілених і гетерогенних програмних систем. Структура розроблена як співпраця децентралізованих автономних менеджерів (АМ) (рис. 2.1) , які здатні до самоорганізації для досягнення спільної мети. Автономні менеджери керуються архітектурою в тому сенсі, що вони переслідують цілі, виражені у вигляді абстрактних архітектурних моделей; і підтримувати відомості про керовані ресурси у формі архітектурних моделей часу виконання.

Точніше, автономні менеджери керуються архітектурною специфікацією під назвою Архетип. Архетип представляє цілі проектування системи, які повинні бути задумані таким чином, щоб гарантувати бажані властивості системи (наприклад, функціональність і атрибути якості). Архетип накладає низку архітектурних обмежень, які повинні бути виконані в будь-який момент часу, водночас залишаючи певний простір для архітектурних варіацій і, отже, адаптації під час виконання. Це абстрактна специфікація, де архітектурні обмеження виражені в термінах типів елементів (а не конкретних реалізацій або екземплярів) і зв’язків; і анотований додатковими властивостями.

Використання архетипів для вираження цілей управління дозволяє нам мати справу з двома цілями дизайну, які, здавалося б, протилежні. По-перше, ми хотіли б, щоб автономна система могла адаптуватися до непередбачуваних ситуацій, як-от робота з нестабільними клієнтськими навантаженнями, оновлення реалізацій внутрішніх компонентів або різноманітні конфігурації - основної розподіленої платформи. У той же час ми хочемо переконатися, що основні властивості системи завжди будуть досягнуті, незважаючи на постійні

зміни та адаптації. Зокрема, такі властивості включають ключові функціональні можливості системи та мінімальну якість обслуговування (QoS).

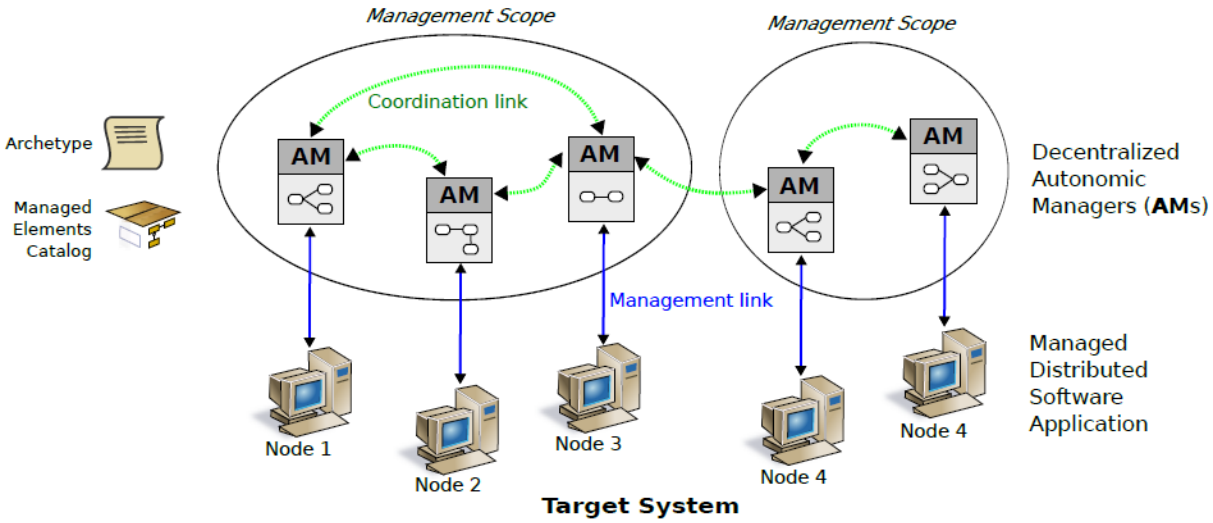


Рисунок 2.1 - Огляд рішення Cube

Архетип забезпечує правильні засоби вирішення обох цих проблем. З одного боку, він визначає жорсткий архітектурний шаблон, який повинен бути реалізований у всіх керованих програмах. Системні адміністратори повинні визначити цю частину, щоб забезпечити основні характеристики системи. З іншого боку, Архетип також визначає ряд точок мінливості, які забезпечують можливість адаптації системи (обмеженої чи безмежної). Системні адміністратори повинні визначити цю частину Архетипу, щоб забезпечити достатні засоби адаптації системи для вирішення ситуацій, що виникають під час виконання. Архетип визначає зону життєздатності керованої системи. Усі керовані програми в стійках мають бути в межах зони життєздатності. Це роль автономного менеджера – підтримувати керовану систему в межах її життєздатності.

Коли мова заходить про розробку частини автономного керування для фреймворку, ми стикаємося з двома додатковими цілями дизайну, які також

здаються протилежними. З одного боку, вимоги до масштабованості та надійності вимагають високого ступеня децентралізації логіки автономного керування. З іншого боку, забезпечення глобально узгодженої поведінки та здатність надійно досягати системних цілей, здається, сприяє суворішому контролю та вищому ступеню централізації. Ми вирішуємо цю проблему, запроваджуючи децентралізовані автономні менеджери для роботи з першим аспектом (масштабованість і надійність); а також цільові механізми самоорганізації та координації для забезпечення другого (когерентність та глобальні властивості).

2.2 Життєвий цикл керування системою

На рис. 2.2 зображено чотири основні види діяльності, задіяні в нашому підході до системного самоуправління. Перші дві дії виконуються вручну та в автономному режимі; вони включають вилучення та специфікацію знань, що стосуються конкретної програми, а також написання та компіляцію коду. Дві дії, що залишилися, автоматизовані та виконуються під час виконання; вони покладаються на процес керування, представлений вище, і не потребують втручання людини або не потребують його.

Фактичні кроки представлені тут після. Метою першого кроку є моделювання системних елементів, які повинні бути самокерованими (наприклад, компоненти програмного забезпечення, конектори, платформи виконання та сервери). Ці елементи моделюються за допомогою мови метамоделювання архітектури , яка постачається разом зі структурою Cube. Основна структура Cube визначає чотири предметно-спеціальні змодельовані елементи: Component , Node і Scope . Компоненти являють собою керовані прикладні модулі, що забезпечують різні функції обробки. Компоненти розгортаються на розподілених платформах (або вузлах). Вузли організовані в різні області (або групи).

Тут ми зауважимо, що адміністратори можуть моделювати інші конкретні системні елементи на основі метамоделі Cube або шляхом розширення тих, що надаються в ядрі інфраструктури. Наприклад , якщо націлено на конкретну область застосування, концепція основного компонента може бути недостатньою для визначення всіх важливих аспектів, якими потрібно керувати в компоненті програми. У цьому випадку адміністратор може розширити артефакт основного компонента та ввести новий, який надає додаткову інформацію.

Крім того, під час першого кроку, разом із змодельованими керованими елементами та їхніми зв'язками, адміністратори повинні впровадити необхідні технологічно-специфічні розширення (TSE) (якщо це ще не зроблено).

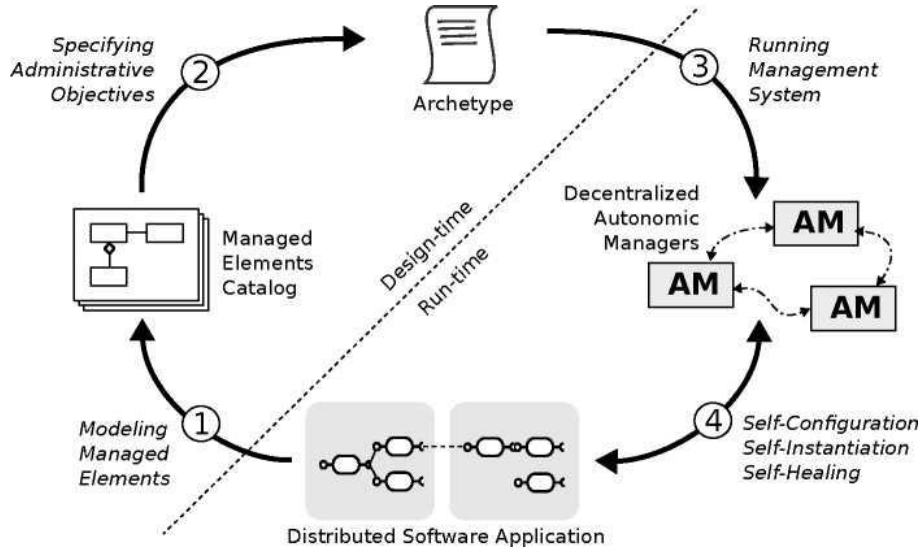


Рисунок 2.2 - Життєвий цикл процесу автономного керування на основі Cube

Вони повинні відповідати конкретним технологіям, що реалізують функції моніторингу та виконання, специфічні для елементів, що представляють інтерес у середовищі виконання.

Під час другого кроку адміністратори вказують цілі самоуправління як набір архітектурних обмежень і атрибутів якості для змодельованих керованих

елементів. Для цього вони спираються на моделі, визначені на першому кроці. Таким чином, цілі управління формально визначені у формі декларативної архітектурної специфікації (архетипу). У рамках цієї дисертації ми припускаємо, що Архетип є достатньо абстрактним, щоб не потребувати подальшого оновлення протягом життєвого циклу системи.

Третій крок полягає в розповсюдженні Архетипу всім автономним менеджерам і завантаженні рівня керування. Таким чином, кожен автономний менеджер отримує копію Архетипу. Після ініціалізації автономні менеджери співпрацюють, щоб сформувати області керування (як визначено в Архетипі).

Нарешті, під час четвертого кроку кожен автономний менеджер відстежує свою частину керованої програми, вирішує локальні проблеми та координує роботу з іншими автономними менеджерами (локально або на рівні Scope) для вирішення проблем нелокального адміністрування. На цьому етапі система C ube працює автономно і робить все можливе для пошуку рішень для всіх адміністративних завдань. Автономний менеджер може бути як реактивним – лише виконувати у відповідь на зміни в керованій системі; і проактивний - накладення архітектурних конфігурацій на частину керованої програми, щоб відповідати цілям, визначеним в Архетипі.

Cube потребує моделювання для двох конкретних цілей. По-перше, Cube формалізує цілі керованої системи через архетип, який набуває форми абстрактної архітектурної моделі. По-друге, Cube представляє стан керованої системи через модель часу виконання, яка приймає форму конкретної архітектурної моделі. Під час виконання роль функцій автономного керування Cube полягає в тому, щоб гарантувати, що модель середовища виконання системи відповідає архетипу. У цьому розділі ми зосереджуємося на мовах моделювання, які надає фреймворк Cube для реалізації цього підходу. Для ясності тут ми робимо абстракцію щодо децентралізації автономних менеджерів і того факту, що модель часу виконання системи фактично розділена між ними. Представлені мови та принципи застосовуватимуться без змін до децентралізованого випадку.

Два типи моделей - архетип і модель часу виконання - вимагають різних мов визначення. Фреймворк Cube визначає дві такі формальні мови:

- System modeling language (SML) - для моделювання предметно-орієнтованих елементів;
- Мова опису цілей (GDL) - для опису цілей користувача в архетипі.

У наступних підрозділах ми детально описуємо ці два типи мов і пояснюємо тісний зв’язок між специфікаціями, які вони створюють. Коротше кажучи, експерти галузі визначають конкретні керовані елементи на основі мови моделювання системи (SML). Отримані специфікації елементів потім використовуються для представлення стану системи під час виконання в моделі середовища виконання Cube. Ця модель представляє основні системні знання, які Autonomic Manager використовує для адміністрування системи. Системні адміністратори використовують мову опису цілей (GDL) для визначення цілей управління в архетипі Cube. Цілі визначаються як формальні обмеження щодо архітектури системи або властивостей якості. Зокрема, на цьому етапі адміністратори можуть використовувати специфікації керованого елемента, визначені раніше через SML, як параметри для обмежень керування, визначених через GDL.

Розробляючи мови специфікації Cube для визначення цілей і моделювання системи, ми врахували такі міркування:

- Має бути можливість моделювати будь-яку частину керованої системи, а також будь-які зв’язки між цими частинами;
- Екземпляри змодельованих системних елементів є представленнями реальних елементів керованої системи. Щоб мінімізувати розмір моделі та спростити її інтерпретацію, екземпляри змодельованого елемента повинні відображати лише властивості, якими ми хочемо керувати, а не всі деталі фактично керованого елемента.
- Змодельовані екземпляри елементів мають бути узгодженими – усі вони мають мати одну мета-модель, незалежно від того, що вони представляють;

- Цілі мають бути визначені безпосередньо на змодельованих елементах системи простою зв’язною та інтерпретованою мовою.
- Під час виконання ми повинні мати можливість переходити між змодельованими екземплярами елементів, перевіряти їх, щоб дізнатися про стан системи, і змінювати їх для адаптації системи.

Нижче ми спочатку представляємо деякі ключові концепції, пов’язані з моделлю, які потім використовуємо для деталізації цілей і специфікацій моделювання системи, запропонованих у структурі Cube.

Завантаження та використання моделей під час виконання системи є частиною відносно недавньої дослідницької ніші, де моделі використовуються не лише під час проектування, але також стають доступними під час виконання, щоб надати більше знань про систему. Слідуючи цій ідеї, кілька фреймворків автономних обчислень запропонували використовувати моделі як основу для механізмів самоадаптації [9]. Тут різні мови моделювання найкраще підходять для різних типів завдань [6]. Архітектура програмного забезпечення розглядається як особливий вид моделі, яка найбільше підходить для представлення програмного забезпечення у вигляді абстрактних функціональних блоків та їх взаємозв’язків. Cube дотримується цього підходу, поміщаючи архітектурні моделі в основу свого управлінського рішення.

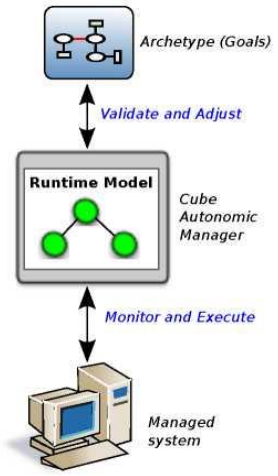


Рисунок 2.3 - Огляд керування Cube на основі моделі

Точніше, щоб керувати компонентними програмними системами, Cube покладається на два типи архітектурних моделей. По-перше, абстрактна архітектурна модель - тип arche - визначає цілі управління. По-друге, конкретна архітектурна модель - модель часу виконання - описує стан системи під час виконання.

Autonomic Manager Cube постійно порівнює модель середовища виконання з архетипом, перевіряє, чи стан системи відповідає цілям управління, і вживає заходів для адаптації системи, якщо це не так. Отже, архетип представляє цілі автономного менеджера. Модель часу виконання представляє свої системні знання, необхідні для досягнення цих цілей. Те ж саме стосується і децентралізованого випадку, коли кожен автономний менеджер виконує описану вище процедуру для окремої частини керованої системи та відповідної частини архетипу.

2.3 Мова моделювання системи та дослідження метамоделі куба

Щоб дозволити розробникам зробити свої системи самокерованими за допомогою фреймворку Cube, перше, що потрібно зробити, це змоделювати елементи системи, щоб ними могли керувати автономні менеджери Cube. Для цієї мети платформа Cube надає мову моделювання системи (SML). Розробляючи SML, ми хотіли досягти двох речей. По-перше, ми хотіли покластися на існуючу мову моделювання, таку як UML або ECORE, щоб отримати вигоду від досвіду, закладеного в її дизайні. По-друге, ми хотіли мати можливість моделювати конкретні системні елементи, такі як ті, що потрібні для підходу Cube і різних областей застосування. Щоб досягти обох цих цілей, ми представили багатошаровий дизайн для підтримки системного моделювання Cube (рис. 2.2). Тут модель в одному шарі повинна відповідати абстракціям моделювання, визначеним у шарі вище. Ми можемо сказати, що модель визначена на основі мови, наданої на рівні вище.

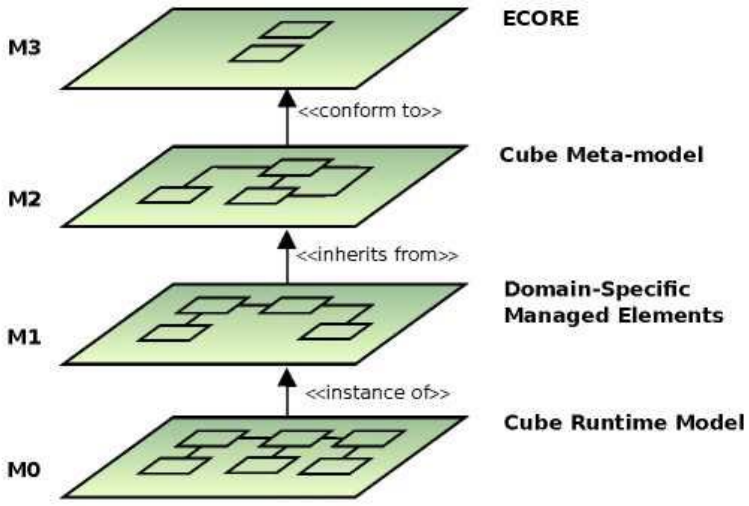


Рисунок 2.4 - Рівні та мови моделювання фреймворку Cube

Коротше кажучи, мова моделювання ECORE була прийнята для представлення найвищого загального рівня (M3). Цю загальну мову можна використовувати для моделювання елементів, які є специфічними для підходу Cube, у шарі нижче (M2). Далі нижче мова моделювання системи (SML) розширює рівень моделювання куба за допомогою предметно-спеціальних елементів (M1). Рівень SML розташований над рівнем моделювання часу виконання (M0), тобто SML використовується для опису елементів моделі середовища виконання. Модель середовища виконання представляє самий нижній рівень. У решті цього підрозділу ми надаємо більш детальну інформацію про складові шарів системного моделювання, запропонованих у структурі Cube.

Фреймворк Cube визначає концепцію «керованого елемента» для представлення будь-якого керованого ресурсу в цільовій системі. Цю загальну концепцію можна розширити, щоб представити більш конкретні компоненти системи, якими ми хочемо керувати, починаючи від детальних програмних компонентів і закінчуючи великомасштабними програмними системами, такими як бази даних або сервери додатків. Концепція керованого елемента визначена на рівні M2 - «Мета-модель куба». Цей рівень визначається за допомогою мови ECORE (або метамоделі). ECORE (рівень M3) є частиною Eclipse Modeling

Framework (EMF). Щоб відповідати цій вимозі, також можна використовувати інші рамки метамоделювання.

Рівень метамоделі Cube (M2) є найважливішим шаром у стеку моделювання. Потім його можна розширити на рівні M1 нижче за допомогою «специфічних для домену керованих елементів», щоб визначити більш точні абстракції фактичних керованих елементів, таких як компоненти, вузли тощо. Під час виконання Auto nomic Managers створюють і підтримують екземпляри таких предметно-спеціальних елементів моделі в моделі середовища виконання, щоб представляти реальні керовані елементи . Це відповідає шару M0 - «Модель виконання куба». Цей рівень може включати кілька змодельованих екземплярів предметно-специфічних керованих елементів - наприклад, кілька екземплярів одного і того ж змодельованого компонента або кілька екземплярів різних змодельованих компонентів. Проте всі змодельовані екземпляри мають однакову внутрішню структуру даних, яка відповідає рівню M2 — мета-моделі куба — навіть якщо вони є екземплярами елементів моделі, визначених на рівні Ml — доменно-специфічні керовані елементи. Дійсно, елементи моделі на рівні M2 успадковують концепцію керування елементами рівня M2.

Усі шари моделювання обробляються фреймворком Cube і реалізовані в прототипі. Єдиний рівень, який розширюють користувачі, — це M1 — «Доменно-специфічні керовані елементи» — для моделювання окремих елементів у кожній керованій системі.

Щоб визначити метамодель Cube (M2), нам потрібна мова метамоделювання (M3) , яка дозволить нам написати специфікацію моделі, а також допоможе нам створити код безпосередньо з розробленої моделі. Незважаючи на існування кількох систем моделювання, таких як MOF ми вирішили використовувати ECORE як основу для визначення метамоделі Cube. Це пояснюється доступністю пов’язаних інструментів, які спрощують використання та маніпулювання концепціями мета-моделі та забезпечують можливість генерації коду. ECORE є частиною проекту Eclipse Modeling Framework (EMF) [13]. EMF — це структура моделювання та засіб генерації коду

для створення інструментів і програм на основі структурованої моделі даних. Дані модель визначена в XMI (стандарт для обміну інформацією метаданих через XML) і відповідає метамоделі ECORE. EMF надає інструменти для створення набору класів Java для моделі, набору класів адаптерів, які дозволяють переглядати, редагувати модель на основі команд і базовий редактор. Він також забезпечує підтримку під час виконання для моделей ECORE, включаючи сповіщення про зміни, підтримку постійності з серіалізацією XMI за замовчуванням і дуже ефективний рефлексивний API для загального маніпулювання об’єктами EMF.

Метамодель ECORE включає наступні концепції (рис. 2.5) :

- EClass : представляє клас з нулем або більше атрибутами та нулем або більше посиланнями;
- EAttribute : представляє атрибут, який має назву та тип;
- EReference : представляє один кінець асоціації між двома класами. Він має прапорець, щоб вказати, чи він представляє вміст і еталонний клас, на який він вказує;
- EDataType : представляє тип атрибута, наприклад int , float або java.util.Date.

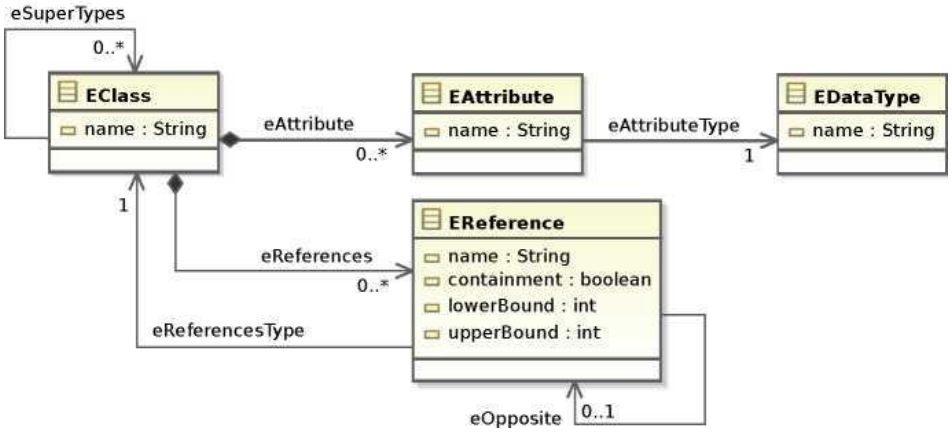


Рисунок 2.5 - Мета-модель ECORE

Рівень метамоделі Cube (M2) використовує ECORE для визначення змодельованих елементів, які є специфічними для підходу Cube. Ми пропонуємо

абстрактну модель (або мета-модель), для представлення будь-якого керованого елемента, який обробляється фреймворком Cube. Мета-модель Cube визначає такі концепції.

• ManagedElement: представляє будь-який керований елемент системи. Його атрибути перераховані нижче. Деякі з цих атрибутів, наприклад ім’я та простір імен, слід вказати під час налаштування класу ManagedElement (у M2) для представлення доменно-спеціального керованого елемента (у M1). Інші атрибути

- ім'я : ім'я керованого елемента, наприклад Component, Node тощо;
- простір імен : представляє контейнер для набору керованих елементів;
- uuid : універсальний унікальний ідентифікатор керованого елемента;
- am : URI автономного менеджера, де розміщено цей керований елемент;
- стан : представляє стан виконання екземпляра цього керованого елемента.

• Властивість : представляє будь-яку функціональну або нефункціональну властивість, яку ми хочемо враховувати як частину операцій самоуправління керованого елемента. Це може включати, наприклад, значення стану, як-от споживання процесора чи оперативної пам’яті, або значення конфігурації, як-от максимальні вхідні дані сервера. Властивість має назву та значення.

• Посилання: представляє будь-який можливий зв’язок між двома керованими елементами. Будь-який новий доменно-спеціальний керований елемент, який ми моделюємо на рівні нижче (M1), повинен додати всі свої зв’язки з іншими керованими елементами як посилання. Не допускаються прямі посилання на Java. Зауважте, що посилання повинно мати назву. Крім того, посилання може бути унарним – це означає, що поточний керований елемент може мати щонайбільше один елемент, на який посилається це посилання; або

множинний — це означає, що керований елемент може мати кілька посилань з однаковою назвою (наприклад, набір вихідних компонентів одного компонента).

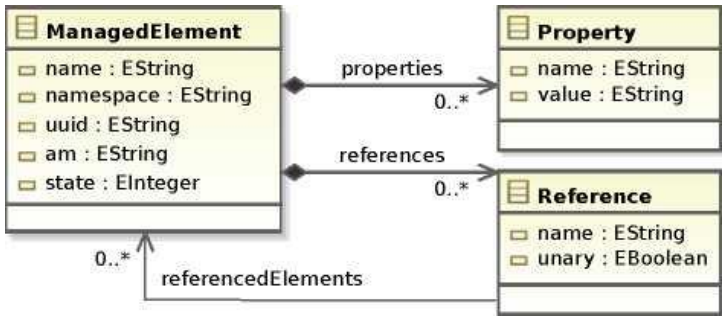


Рисунок 2.6 - Мета-модель куба

Як ми вже згадували раніше, ми використовували ECORE для визначення нашої мета-моделі Cube. Рис. 2.6 деталізує приклад зв'язку між частиною мета-моделі Cube та її відповідним екземпляром моделі ECORE. Тут клас ManagedElement у мета-моделі Cube відповідає об'єкту класу EClass в екземплярі моделі ECORE, для атрибута «ім'я» якого встановлено значення «ManagedElement». Атрибути ManagedElement відповідають об'єктам класу EAttribute ECORE. У прикладі, показаному на Рис., ми показуємо лише два атрибути — «name» та «namespace» — які мають тип ESTring. Концепція властивостей метамоделі Cube відповідає іншому об'єкту EClass метамоделі ECORE , з атрибутом «ім'я» встановленим значенням «Властивість». Асоціація «властивостей» між концепціями ManagedElement і Property на рівні M2 відповідає об'єкту класу EReference ECORE.

Використання EMF дозволило нам створити код Java безпосередньо з мета-моделі Cube, оскільки це було визначено на основі мета-моделі ECORE. Згенерований код має дуже цікаві функції, включаючи механізм сповіщення про зміни та серіалізацію. У нашій реалізації прототипу ми дещо змінили згенерований код, щоб він відповідав нашим потребам Framework. Зокрема, ми опустили прямі посилання Java між змодельованими екземплярами елементів (наприклад, посилання між змодельованими об'єктами). Замість цього ми

використали універсальний унікальний ідентифікатор (UUID) для представлення посилань між різними змодельованими примірниками. Це дозволяє переміщувати змодельовані екземпляри між різними віртуальними машинами Java без зміни всієї моделі.

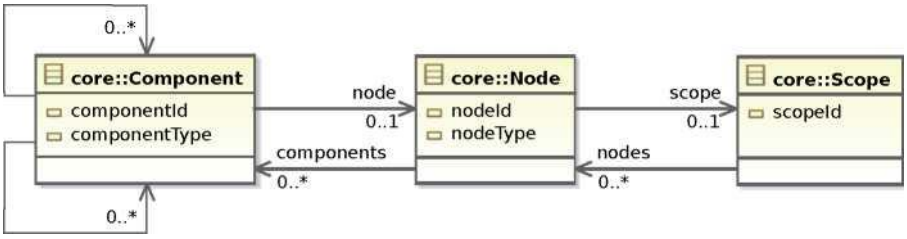


Рисунок 2.7 - Фрагмент метамоделі куба та еквівалент екземпляра моделі ECORE

Оскільки фреймворк Cube націлений на область посередництва, ми запропонували «основну» предметно-спеціальну модель для абстрагування та представлення систем посередництва. У наступному описі та решті цієї дипломної роботи буде використано цю основну модель для опису та пояснення різних частин фреймворку.

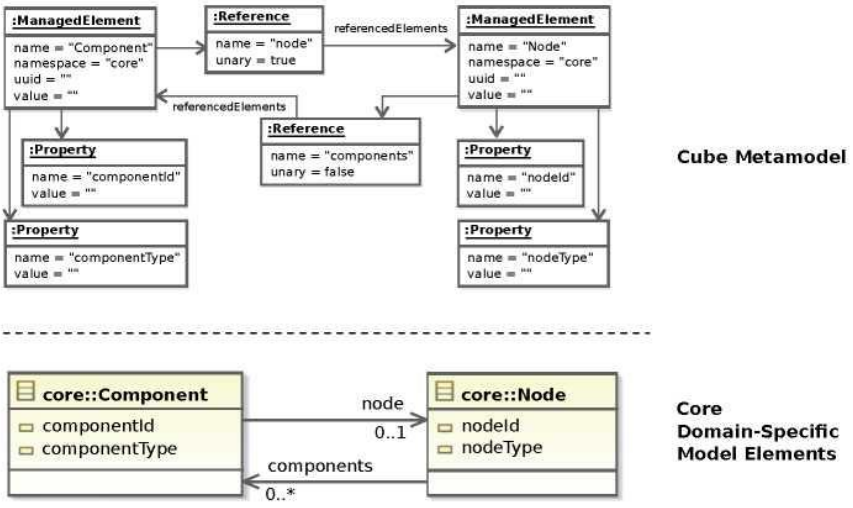


Рисунок 2.8 - Основні доменно-специфічні керовані елементи фреймворка Cube

На рис. 2.8 показані основні доменно-спеціальні керовані елементи, які ми пропонуємо. Ми детально описуємо кожен елемент у наступному списку:

- Компонент : представляє абстракцію програмного компонента. Має дві властивості: componentId - ідентифікатор екземпляра компонента; і componentType - тип компонента. Ці дві властивості встановлюються під час виконання під час створення екземпляра елемента моделі Component. Компонент має три посилання: входи та виходи, які представляють вхідні та вихідні посилання на інші компоненти; і посилання на вузол, яке містить UUID вузла, де розміщено компонент.
- Вузол : представляє вузол – платформу розгортання або фізичний пристрій – де можуть виконуватися екземпляри компонентів. Node має дві властивості: nodeld , яка містить ідентифікатор екземпляра Node; і nodeType, який містить тип вузла, наприклад «ПК», «Смартфон» тощо. Елемент Node має два посилання: компоненти, які містять список посилань на розміщені компоненти; і область, яка містить посилання на область, членом якої є вузол.
- Область : представляє віртуальне групування вузлів. Наприклад, він може представляти домен адміністративної мережі, географічне розташування або набір платформ, що мають певні характеристики. Елемент Scope має одну властивість: scopeld , яка містить ідентифікатор області. Він також має два посилання: вузли, які містять список членів області; і головний (не показаний на Рис.), який містить посилання на головний лідер області. Створений екземпляр елемента моделі Scope представляє провідну область цієї області.

2.4 Дослідження внутрішньої архітектури автономного менеджера

Щоб автоматизувати завдання керування програмними системами, нам потрібна формальна мова для визначення цілей адміністрування з точним, але простим синтаксисом. У цій дипломній роботі ми пропонуємо мову опису цілей (GDL) для визначення цілей у структурі Cube. Це формальна мова для визначення

архетипу Куба.

GDL дозволяє системним адміністраторам вказувати цілі управління як обмеження для елементів керованої системи. Тут керовані елементи, змодельовані раніше за допомогою SML, можна використовувати як параметри для обмежень GDL. Отримана специфікація — архетип Cube — надається як вхідна інформація для всіх автономних менеджерів Cube, які беруть участь в адмініструванні системи. Ця вхідна специфікація представляє цілі адміністрування, яких повинні досягти автономні менеджери.

За своєю суттю GDL є формальною моделлю для опису адміністративних цілей, які повинні бути досягнуті на наборі керованих елементів. Таким чином, GDL може описати керовані елементи, націлені на самоуправління, і визначити цілі, яких потрібно досягти на цих елементах. Цілі можуть бути виражені в термінах властивостей, які повинні бути досягнуті на одному керованому елементі, або в термінах посилань, які повинні бути забезпечені між двома керованими елементами. Приклади включають два компоненти певного типу, які завжди повинні бути з’єднані; або максимальну прийнятну кількість клієнтів сервера. Звичайно, спосіб досягнення таких цілей може залежати від розподіленого середовища виконання.

Синтаксис GDL можна абстрагувати у вигляді простої моделі на основі графів. Це дає змогу просто представити цілі та твердження щодо керованих елементів у вигляді спрямованого графа. Тут вершини представляють керовані елементи, а ребра представляють їхні цільові властивості або відносини між ними (наприклад, посилання на інші елементи). Основна мета моделі GDL полягає в тому, щоб дозволити адміністраторам визначати цілі самоуправління з точки зору обмежень, яким система повинна відповідати під час свого виконання. Ця мета перетворюється на визначення достатньо детальних описів керованих елементів і властивостей або посилань, яким повинна задовольняти структура Cube в будь-який момент часу.

На рис. 2.9 показано графічне представлення прикладу моделі GDL. У цьому прикладі ми хочемо гарантувати, що будь-який компонент типу «Decoder»

підключено до іншого компонента типу «MailSender-Компонент», який має бути на вузлі типу «MailServer». Отже, мета цієї специфікації полягає в тому, щоб ці два компоненти завжди були з’єднані. Мета всіх інших наданих деталей полягає в тому, щоб точно визначити, які компоненти підпадають під цю мету самоконтролю.

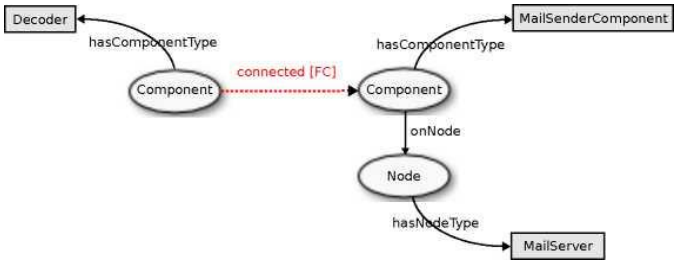


Рисунок 2.9 - Приклад GDL

Загалом GDL базується на ідеї створення тверджень про керовані елементи у формі виразів суб’єкт-ціль-об’єкт. Тут предмет представляє точний опис керованого елемента з точки зору його властивостей, значень властивостей і посилань. Об'єкт - це опис іншого керованого елемента, який пов'язаний з предметним елементом (через мету). Ціль виражає характеристику (характеристики), якою предметний елемент повинен володіти в будь-який момент часу. В обмеженні GDL, наведеному вище, компонент «Decoder» представляє суб’єкт, «MailSenderComponent» — об’єкт, а «connected» — ціль. Це вказує, що будь-який суб’єктний компонент має бути підключений до об’єктного компонента.

Якщо об’єкт є літеральним значенням, метою є керування властивістю суб’єкта керованого елемента (наприклад, діапазон значень для атрибута компонента). Якщо об’єкт представляє інший керований елемент, мета визначає двійкову властивість, яка повинна бути забезпечена між двома елементами (наприклад, два компоненти, які повинні бути з’єднані). Зауважте, що вираз суб’єкт-ціль-об’єкт також використовувався в інших моделях, наприклад у Resource Description Framework (RDF), але на основі іншого шаблону – суб’єкт-предикат-об’єкт – і для іншої ролі – призначеної для опису Інтернету. Ресурси.

Основна відмінність полягає в тому, що RDF був призначений для визначення статичних веб-ресурсів, щоб полегшити автоматичну обробку їх описів; у той час як GDL призначений для визначення цілей користувача з детальним описом архітектури системи та складових керованих елементів.

GDL призначений для автоматичної обробки фреймворком Cube, а не лише для відображення для адміністраторів. Крім того, модель заснована на абстрактній архітектурній моделі, а не на прямому вираженні та описі елементів керованої системи. А саме, вершини в графі GDL є простими описами керованих елементів, які повинні бути частиною цілі самоуправління (наприклад, компонент типу «Декодер»). Вони не є конкретними специфікаціями, що стосуються унікального керованого елемента (наприклад, не повний ідентифікатор конкретного класу Java або реалізації Компонента від певного постачальника).

Мова GDL спирається на три основні концепції:

- Елемент (керований або примітивний);
- Власність (ціль або опис);
- Архетип.

Елемент представлений вершиною в графі GDL. Він може представляти дві речі:

1. Керований елемент (ME): представляє опис керованого елемента, що підлягає цілям(ам) самоуправління. Він може представляти будь-який цільовий керований елемент, який уже був змодельований на основі мови моделювання системи (наприклад, компонент, вузол і область).

2. представляє постійний літерал, визначений через значення рядка.

У представленні GDL на основі графів властивості зображуються як ребра (або дуги) і позначаються назвою властивості. Ребра спрямовані від суб’єкта до об’єкта, іншими словами, від елемента, якого стосується властивість, до елемента, який надає більше інформації про властивість. Нарешті, властивості Опису зображені суцільними чорними краями, а властивості Цілі – пунктирними червоними краями.

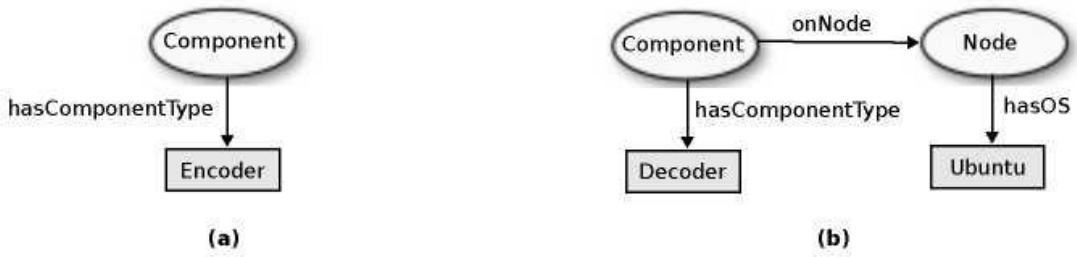


Рисунок 2.10 - Приклади властивостей опису в моделі GDL

На рис. 2.10 показано два приклади властивостей GDL, які використовуються як описи (суцільні темні краї). На Рис. 2.10a зображено унарну властивість. Керований елемент типу «Component» (суб'єкт) має унарну властивість «hasComponentType», де об'єкт є примітивним елементом, якому присвоєно значення рядка «Encoder». Ця проста властивість Опису інтерпретується наступним чином: «будь-який компонент, який має тип Encoder». На Рис. 2.10b зображено двійкову властивість. Тут компонент описується як такий, що має тип «Декодер», а також як розгорнутий на вузлі, який має операційну систему (ОС) «Ubuntu». Цю другу специфікацію GDL можна інтерпретувати наступним чином: «будь-який компонент типу Decoder, який знаходиться на вузлі з операційною системою Ubuntu». На даний момент мета ще не визначена. Це лише описи, які можна використовувати пізніше для визначення фактичних цілей (Властивості цілі).

Коли властивості GDL використовуються як цілі (пунктирні, червоні дуги), вони представляють мету, яку потрібно досягти. На рис. 2.11 пов'язана властивість вказана як мета, яку потрібно досягти. Тобто Компонент типу «Кодер» має бути підключений до іншого Компонента типу «Декодер», який знаходиться на вузлі з ОС «Ubuntu».

Порівняно з властивостями опису, властивості цілі надають додаткову інформацію про стратегію вирішення проблеми автоматичного менеджера. Ця інформація корисна для оптимізації процесу вирішення. У Cube, намагаючись досягти цілей користувача, автономний менеджер використовує Archetype

Resolver – внутрішній модуль. Важливим аспектом тут є те, що Archetype Resolver будує внутрішній графік роздільної здатності на основі архетипу та намагається знайти конфігурації моделі середовища виконання, які задовольняють цільові властивості та описи на графіку. Розглядаючи приклад, зображений на рис. 2.11, архетип намагається досягти мети, пов'язаної з суб'єктом, який є Компонентом типу «Кодер». Щоб досягти цієї мети, Archetype Resolver спочатку намагається знайти цільовий об'єкт, який є компонентом типу «Декодер» і розташований на вузлі, який має ОС «Ubuntu».

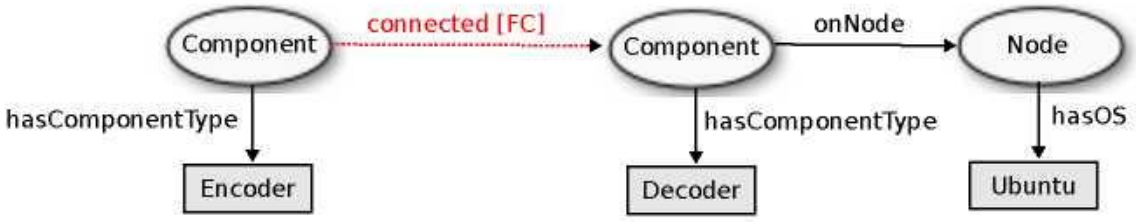


Рисунок 2.11 - Приклад властивості цілі в моделі GDL

Пошук виконується на основі інформації, доступної в моделі виконання. Якщо екземпляр компонента, який відповідає цьому опису, не знайдено, Archetype Resolver створює змодельований екземпляр компонента з цим описом і розміщує його в моделі середовища виконання. Ця поведінка пошуку (F) або створення (C) явно вказана у зв'язаній меті через літерали (FC).

Archetype Resolver, розроблений як частина прототипу фреймворку Cube, підтримує наступні стратегії вирішення:

- FIND [F]: знайти змодельований екземпляр елемента, який відповідає опису, наданому елементом object бінарної властивості.
- FIND_OR_CREATE [FC]: почніть із пошуку змодельованого екземпляра елемента, який має заданий опис; тоді, якщо не знайдено, створіть цей екземпляр у моделі середовища виконання.
- CREATE [C]: безпосередньо створити екземпляр із описом, наданим в елементі object, не намагаючись шукати існуючий екземпляр.

Загалом дизайн Cube Autonomic Manager (AM) відповідає досить «класичному» автономному обчисленню MAPE- K 5 петля, запропонована IBM. Крім того, він також пропонує кілька важливих доповнень. На рис. 2.12 показана внутрішня архітектура автономного менеджера, як це визначено у структурі Cube.

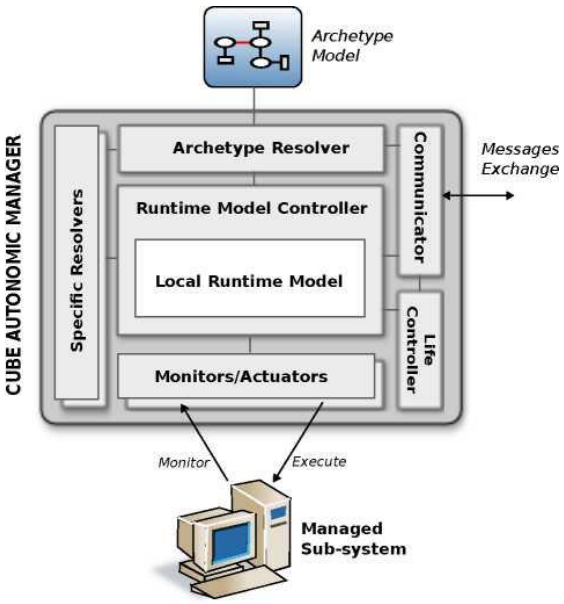


Рисунок 2.12 - Автономний менеджер Cube framework

Модулі моніторингу та виконання в Cube AM виконують ту ж роль, що й у циклі MAPE-K. Подібним чином завдання планування та аналізу в циклі MAPE-K еквівалентні модулю Archetype Resolver від Cube . Archetype Resolver забезпечує функцію прийняття рішень Cube AM. Він базується на системних знаннях, представлених у локальній моделі виконання, і переслідує цілі, визначені в моделі архетипу. Локальна модель середовища виконання представляє часткове уявлення адміністратора про керовану систему, описане через змодельовані екземпляри елементів (або просто змодельовані екземпляри) та їхні взаємозв'язки. Усі змодельовані екземпляри всіх Cube AM складають глобальну модель виконання, яка відображає загальний стан всієї керованої системи. Доступ до локальної моделі середовища виконання для цілей читання та запису забезпечується контролером моделі середовища виконання через

інтерфейс доступу спеціального призначення . Модель Архетипу представляє цілі користувача, яких потрібно досягти.

Якщо мети неможливо досягти локально, Cube AM має співпрацювати з іншими Cube AM. Для цього він покладається на внутрішній комунікаційний модуль для обміну асинхронними повідомленнями з AM, з якими він співпрацює, так звані «сусіди». Після зв'язку Cube AM також перевіряє благополуччя своїх сусідніх AM за допомогою спеціального внутрішнього модуля – Life Controller (або Failure Detector). Метою цього модуля є перевірка стійкості цільових рішень, які були встановлені у співпраці з сусідами - наприклад, екземпляри віддалених елементів, які допомагають розв'язати обмеження підключення, можуть зникнути, коли сусідня AM виходить з ладу.

Щоб забезпечити розширюваність, необхідну для підтримки майбутніх бізнес-цілей і різних типів керованих ресурсів, внутрішня архітектура Cube AM містить два типи модулів: основні модулі та модулі розширення.

Модулі розширення можуть бути додані до Cube AM для збагачення його функцій системними можливостями . Це включає наступні модулі: комунікатор, монітори/виконавці, контролер життя та розв'язувачі цілей. Модулі розширення дозволяють спеціалізувати AM Cube для конкретних випадків керування та середовищ виконання. Усі модулі розширення можна динамічно змінювати, додавати та видаляти за потреби.

Висновки до розділу 2

В даному розділі виконано алгоритмічну та структурну реалізація фреймворку автономних обчислень. Виконано опис проектного рішення, наведено життєвий цикл керування системою та мову моделювання системи та дослідження метамоделі куба.

РОЗДІЛ 3. Імплементація моделей децентралізованої архітектури в систему автономного обчислення

3.1 Процес оцінки та застосування запропонованої методології

Структуру Cube було оцінено за допомогою двох експериментальних прикладів. Перше прикладне дослідження стосується самостійного створення та конфігурації розподіленої компонентної програми (тобто системи посередництва даних). Ми показуємо, як можна використовувати Cube для автоматизації створення та взаємозв'язку примірників компонентів у системах розподіленого програмного забезпечення. Ми також оцінюємо продуктивність і потенціал масштабованості нашого рішення Cube для досягнення таких цілей у порівнянні з ручними або централізованими рішеннями. У другому прикладі ми зосереджуємо процес оцінювання на додаткових, додаткових вегетативних функціях фреймворку Cube. Зокрема, ми показуємо, як фреймворк Cube може забезпечити самовідновлення розподілених програм, коли базові вузли виконання зазнають збоїв під час виконання. Ми також показуємо, як Cube може самостійно оптимізувати споживання ресурсів між серверами виконання, розподіляючи навантаження завдань обробки між вузлами кластера. Отже, цей другий варіант використання більше зосереджується на якісному аналізі нашого фреймворку.

Два приклади оцінювання спрямовані на управління програмами-посередниками даних. Посередництво історично використовувалося для інтеграції даних, що зберігаються в IT-ресурсах, таких як бази даних, бази знань, файлові системи, цифрові бібліотеки або системи електронної пошти Загалом посередництво даних реалізується як програмний рівень між джерелами даних (або ресурсами) і приймачами даних (або програмами) (рис. 3.1) . Це забезпечує взаємодію та своєчасну інтеграцію різнорідних інформаційно-орієнтованих елементів.

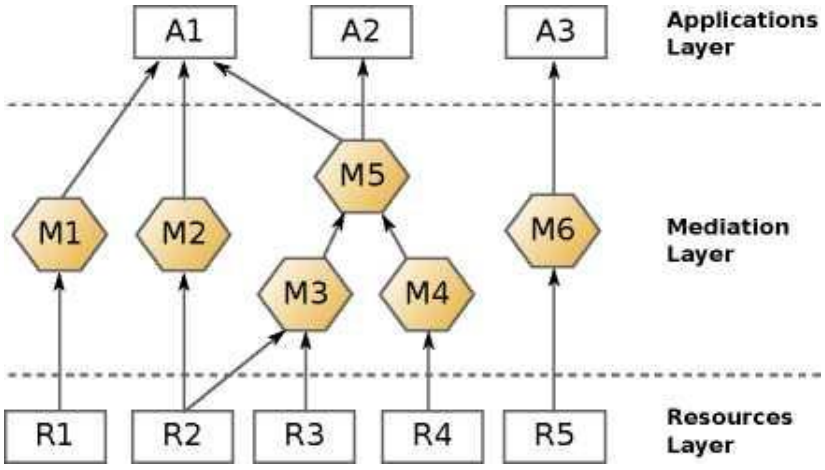


Рисунок 3.1 - Представлення шарів системи

Рівень посередництва складається з набору посередників. Часто визначають посередник як «програмний модуль, який використовує закодовані знання про певні набори або підмножини даних для створення інформації для вищого рівня програм». Таким чином, основна мета рівня посередництва полягає в тому, щоб надати зацікавленим програмам високорівневе представлення даних, отриманих з джерел даних. Загалом, таке представлення «вищого рівня» базуватиметься на концепціях, ближчих до домен цільової програми, ніж концепції «нижчого рівня» домену джерел даних. Для досягнення цих цілей медіатори забезпечують такі операції, як:

- Відбір і фільтрація вхідних даних в залежності від їх характеристик.
- Перетворення між несумісними типами даних.
- Агрегація даних, отриманих із кількох джерел.
- Семантичне вирівнювання неузгоджених даних.

Інкапсуляція таких операцій у спеціалізоване програмне забезпечення для посередництва даних є хорошою практикою. Наприклад, програмне забезпечення-посередник забезпечує єдину точку інтерфейсу для різних програм і ресурсів, які потрібно інтегрувати. Це зменшує кількість необхідних підключень і полегшує керування змінами. Крім того, програмне забезпечення-посередник забезпечує рівень ізоляції від деталей програмного забезпечення,

таких як певні інтерфейси та формати даних. Крім того, якщо його правильно налаштовано, це дозволяє швидко та економічно ефективно розробляти нові інформаційні послуги (наприклад, шляхом реінтеграції джерел даних для отримання нової інформації вищого рівня). Крім того, відокремлюючи джерела даних від приймачів, рівень посередництва покращує можливість повторного використання та еволюцію програм і ресурсів даних. Це також дозволяє прозоро додавати нові властивості QoS (якість обслуговування), такі як безпека, надійність тощо. Нарешті, це може допомогти покращити масштабованість усієї системи.

У наш час сфера застосування підходів, заснованих на посередництві, охоплює багато контекстів застосування, спричинених появою нових середовищ, нових технологій і нових програм. З великої різноманітності таких контекстів найчастіше можна виділити дві основні форми посередництва.

Інтеграція даних: посередники збирають дані з пристроїв; застосовувати операції над зібраними даними, включаючи агрегацію, фільтрацію та кореляцію; і передавати дані в бізнес-додатки або інструменти контролю, узгоджено. Оскільки нові джерела даних можуть з’являтися та зникати в будь-який момент, система посередництва має бути достатньо гнучкою та динамічною, щоб адаптуватись під час виконання без зупинки всієї системи. Цей тип посередництва також відомий як посередництво даних.

Взаємодія додатків: посередники вирішують проблеми взаємодії між різнорідними програмними додатками. У цьому контексті програмне забезпечення-посередник стоїть між клієнтськими програмами та програмами-провайдерами (загалом, веб-службами). Його мета полягає в тому, щоб надати споживачам послуг безперебійний і правильний доступ до наданих послуг. Ця форма посередництва, також відома як Service Mediation, має на меті забезпечити уніфікований або стандартизований інтерфейс для різнорідних послуг. Спочатку він був розроблений спільнотою ESB (Enterprise Service Bus) як невід’ємна частина проміжного програмного забезпечення ESB.

Cilia [9, 10] — це модульний фреймворк для посередництва даних, заснований на технології OSGi та моделі на основі сервісних компонентів iPOJO. Команда Adele була основним розробником цього фреймворку, який наразі використовується в таких спільних ініціативах, як медичний проект. Cilia пропонує як спеціалізовану модель компонентів для систем передачі даних , так і динамічне середовище виконання. Прийняття компонентної моделі має на меті створення посередницьких додатків за допомогою композиції повторно використовуваних компонентів, що забезпечує кращу модульність і гнучкість додатків. Дійсно, на додаток до модульності, динамічність є важливою властивістю фреймворку Cilia, що надає додаткам-посередникам достатню гнучкість для зміни їх архітектури під час виконання.

Технічно кажучи, додаток-посередник Cilia (також званий ланцюгом посередництва) — це набір компонентів посередництва — званих медіаторами — які з'єднані між собою за допомогою посилань — званих прив'язками. Отже, такі додатки-посередники мають форму орієнтованого ациклічного графа (DAG), де вузли є посередниками (отримують, обробляють і передають дані), а ребра є зв'язками між посередниками (передають дані).

Медіатор Cilia має такі внутрішні субкомпоненти (рис. 3.2).

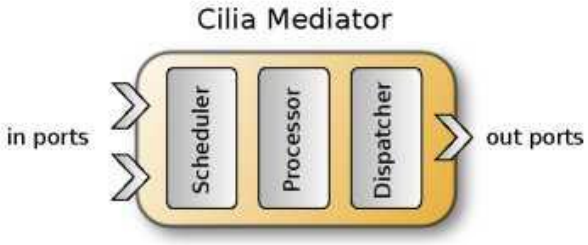


Рисунок 3.2 - Представлення компонента Cilia Mediator та його складові

Планувальник: відповідає за збір вхідних даних з одного або кількох джерел, доки не буде виконано заздалегідь визначену умову . Коли це відбувається, планувальник пересилає всі зібрані дані до процесора посередника (розглянемо далі). Кожна реалізація планувальника забезпечує власну логіку для

відповідної умови запуску. Це може бути часовий, кількісний, якісний і так далі. Деякі типові приклади реалізації планувальника включають: «Періодичний планувальник», який періодично надсилає зібрані дані до процесора; і «Негайний планувальник», який запускає обробку, щойно надходять дані. Дистрибутив Cilia надає кілька попередньо визначених планувальників. Користувачі Cilia можуть розробляти додаткові планувальники або налаштовувати існуючі.

На додаток до медіаторів, описаних вище, Cilia забезпечує особливий вид медіаторів, які називаються адаптерами. Вони розміщуються на кордонах (вхід/вихід) ланцюгів посередництва Cilia, щоб зв’язати їх із різнорідними (не Cilia) джерелами та приймачами даних відповідно. Таким чином, адаптери відповідають за зв'язок із зовнішніми джерелами даних (ресурси) або приймачами (цільові програми). Існує три типи адаптерів: вхідні адаптери, вихідні адаптери та адаптери запит-відповідь.

Посередники (і адаптери) підключаються через прив’язки. Прив'язка описує з'єднання між вихідним і вхідним портами. Поєднані посередники утворюють посередницький ланцюг. Рис. 3.3 ілюструє зразок ланцюга посередництва, що складається з двох адаптерів, чотирьох посередників і шести прив’язок. Cilia надає рефлексивний API, який дозволяє створювати, оновлювати та видаляти посередники і прив’язки з ланцюга посередництва динамічно без зупинки програми.

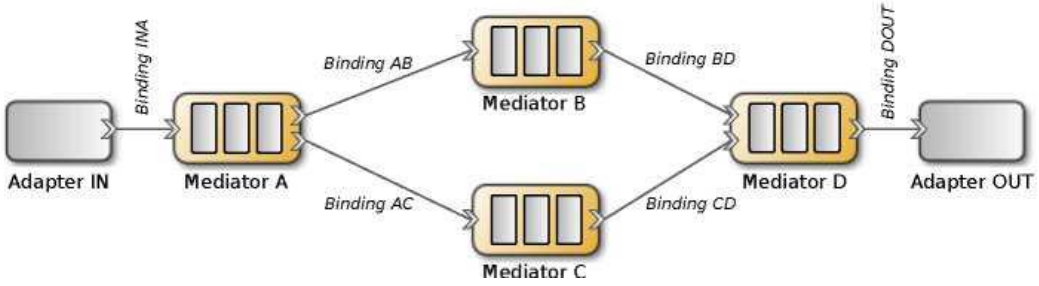


Рисунок 3.3 - Приклад посередницького ланцюга

Цей API експортується як локальна служба OSGi і може використовуватися сторонніми інструментами для маніпулювання ланцюжками посередництва. Ми використовуємо цей API для впровадження розширення фреймворку Cube для Cilia.

Як приклад застосування ми розглянемо систему посередництва для моніторингу споживання домашніх ресурсів. Система вибірки контролює споживання домогосподарства ресурсів, зокрема електроенергії, газу та води. Зібрані дані обробляються для розрахунку різних витрат. На рівні будинку він розраховує витрати на електроенергію, газ і воду. Потім вони об'єднуються у витрати регіону вищого рівня, міста витрати, а потім національні витрати. Відповідні типи компонентів включають спеціальні зонди - збирання вимірювань електроенергії, води та газу; і різні калькулятори витрат - обчислення витрат на споживання на рівні будинку, області, міста та країни.

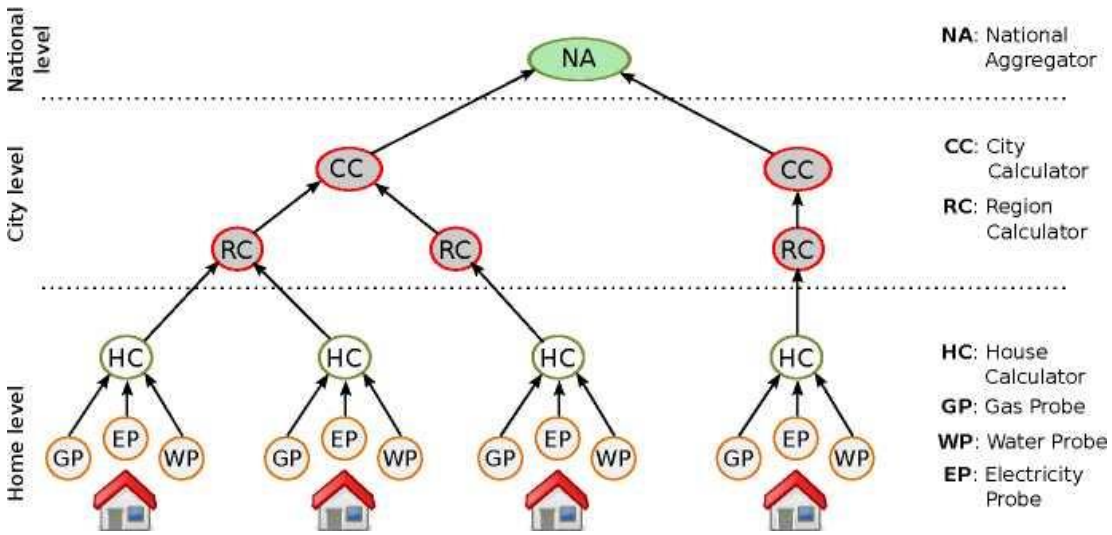


Рисунок 3.4 - Огляд програми моніторингу ресурсів будинку

На рис. 3.4 показано загальну архітектуру нашої розподіленої системи посередництва. Ми розрізняємо три рівні, на яких існують різні види компонентів. Ці рівні були визначені на основі типів залучених адміністративних органів і відповідних базових апаратних платформ.

На нижньому рівні, що представляє будинки, представлено три типи компонентів для збору вимірювань газу, електроенергії та води. Ми називаємо їх газовим зондом (GP), електричним зондом (EP) і водним зондом (WP). Ці компоненти виконуються на спеціальному типі машин, які називаються шлюзами (або домашніми ящиками), доступними в кожному будинку. Кожен із цих компонентів зонда надсилає зібрані дані до компоненту House Calculator (HC), який агрегує дані, обчислює загальну вартість споживання будинку та пересилає їх до регіонального калькулятора витрат.

На цьому проміжному рівні, що представляє цілі міста, існують два види компонентів. Перший, який називається Region Calculator (RC), збирає дані про набір будинків, розташованих в одному районі міста. Отже, для кожного регіону ми маємо один екземпляр цього компонента. Усі екземпляри RG у місті надсилають зібрані дані в унікальний екземпляр на рівні міста, який називається City Calculator (CC). У свою чергу, кожен екземпляр CC аналізує зібрані дані, узагальнює їх і надсилає до національного центру обробки даних (розглянемо далі). На рівні міста у нас є об’єднані серверні машини. Ці машини можуть бути розташовані в одному кластері або розподілені в різних місцях (наприклад, географічно поблизу від контрольованих будинків).

Основна мета цього першого прикладу полягає в тому, щоб оцінити здатність фреймворку Cube самостійно створювати (тобто створювати екземпляри та з’єднувати компоненти) розподілені ланцюжки посередництва, які відповідають специфікації архетипу. Щоб підтвердити цю здатність, медіатори Cilia повинні бути створені автоматично, а їхні взаємозв’язки , налаштовані правильно через структуру Cube. На локальному рівні (в межах однієї платформи OSGi) прив’язки між посередниками Cilia є «прямими» — за допомогою або локальної служби адміністрування подій OSGi, або прямих викликів методів. Однак, перетинаючи межі локальної платформи OSGi, адміністратори використовують адаптери Cilia JMS для передачі повідомлень даних через загальний JM S 2 Сервер (як показано на рис. 3.5) . Зауважте, що використання адаптерів JMS представляє особливий технологічний вибір у представленому

варіанті використання. Інші технології проміжного програмного забезпечення також можуть бути використані для з’єднання розподілених ланцюгів Cilia.

Діяльність за замовчуванням, необхідна для створення екземплярів ланцюжків посередництва Cilia, полягає в написанні XML-файлу DSCilia для кожного локального ланцюга посередництва, який буде виконано на кожній платформі OSGi. Коли файл DSCilia розгортається в каталозі спеціального призначення, «статичний» ланцюжок посередництва, указаний у файлі DSCilia, створюється автоматично (за допомогою Cilia). Щоб з’єднати кілька локальних ланцюжків посередництва та сформувати глобальний розподілений ланцюжок посередництва, адміністратори повинні створити відповідні адаптери JMS на кожній платформі OSGi. Ці адаптери JMS мають бути налаштовані для використання доступного сервера JMS і для спрямування даних до цільового ланцюга(ів) посередництва.

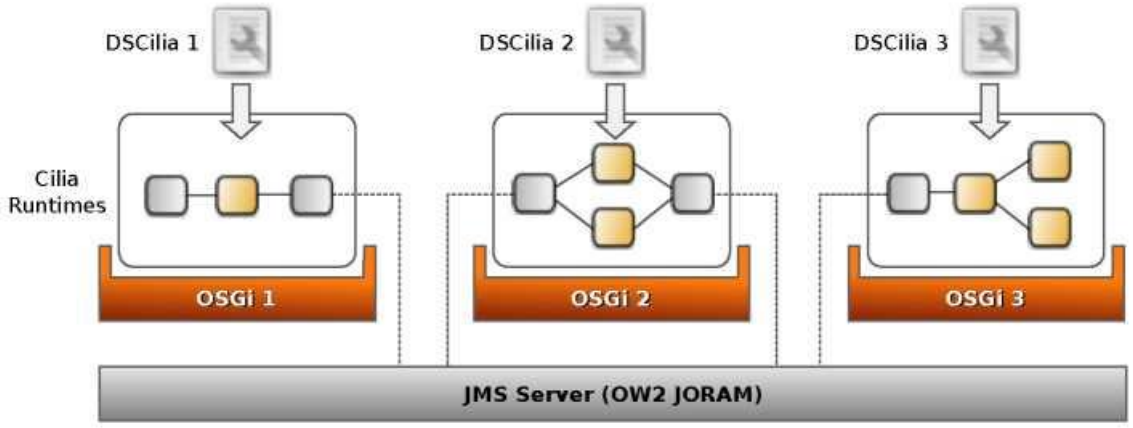


Рисунок 3.5 - Розподілений посередницький ланцюг

Ця операція є відносно простою завдяки можливостям динамічної адаптації Cilia. Однак налаштування всіх локальних ланцюгів посередництва, розташованих у (дуже) великих областях, може швидко стати дорогим і бути схильним до людських помилок.

У цьому розділі ми детально описуємо, як ми використовували структуру Cube для вирішення вищезгаданих проблем. Наш підхід до самоконтролю за

допомогою Cube включає кілька кроків і дій. Перші дві дії виконуються вручну та в автономному режимі. Вони передбачають вилучення та специфікацію знань, пов’язаних із застосуванням. У контексті запропонованої нами структури це відповідає: вибору або визначенню нових абстракцій моделі; і вибір або розробка нових розширень, які надають конкретні компоненти монітора/виконавця та/або конкретні резолвери, пов’язані з керованою системою. Далі користувачі повинні визначити цілі адміністрування на основі мови специфікації Archetype. Отриманий файл A rchetype потім використовується Cube Autonomic Managers для самостійного керування системою та забезпечення досягнення цілей користувача.

Друга частина Архетипу (зображена на рис. 3.6) показує цілі, пов’язані із самостворенням і конфігурацією частини ланцюга посередництва, яка розташована на домашніх вузлах шлюзу. Основні цілі в цій частині Архетипу визначають спосіб, у який Компоненти мають бути взаємопов’язані (за допомогою «підключеної» властивості цілі).

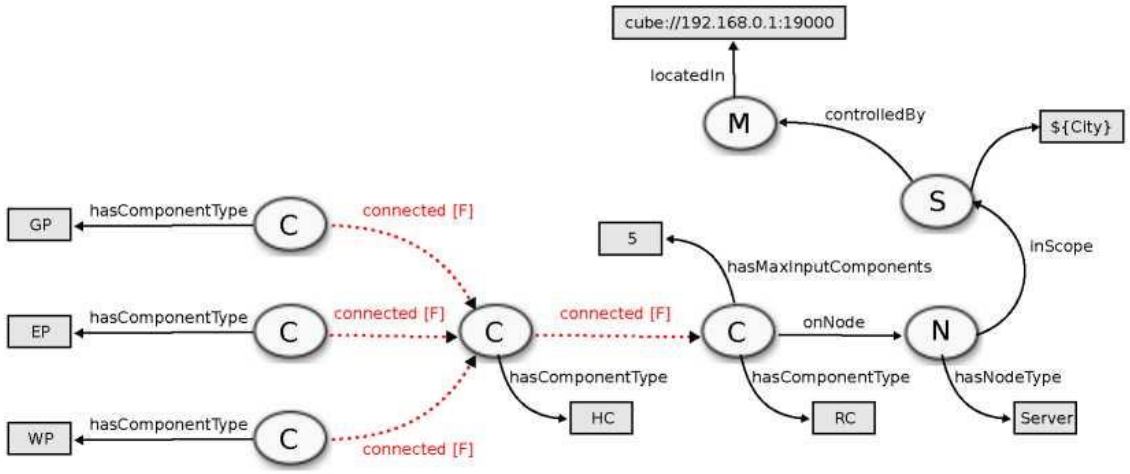


Рисунок 3.6 - Частина архетипу для самостійного створення та підключення компонентів посередництва на домашніх шлюзах

На домашньому рівні потрібно керувати чотирма типами компонентів: газовий зонд («GP»), водяний зонд («WP»), електричний зонд («EP») і

калькулятор вартості будинку («HC»). Екземпляри перших трьох типів компонентів, або зонди, повинні бути підключені до екземпляра компонента «HC». У той же час екземпляр «HC» має бути підключений до віддаленого екземпляра — «RC» (Region Calculator), розташованого на вузлі типу «Сервер» у межах міста цього регіону. Ми також вказали, що знайдений екземпляр “RC” не повинен мати більше п’яти (“5”) вхідних компонентів; це було вказано за допомогою властивості опису «hasMaxInputComponents» компонента «RC». Якщо це не так, Cube має знайти інший екземпляр «RC» на тому самому віддаленому вузлі або на будь-якому іншому вузлі, який відповідає наданому опису (тобто тип вузла «Сервер» і область дії «${City}»).

Третя частина Архетипу (зображена на рис. 3.8) показує цілі, пов’язані із самостійним створенням і налаштуванням частин ланцюга посередництва, які розташовані на міських серверах і національних центрах даних. Тут усі екземпляри компонента «RC» мають бути підключені до екземпляра компонента «CC» (City Calculator), який розташований на одному із Серверів у цьому місті. У свою чергу, будь-який екземпляр «CC» має бути підключений до екземпляра компонента «NA» (національний агрегатор), розташованого в центрі обробки даних.

3.2 Представлення сценаріїв та їх результати імплементації

Ми провели кілька експериментів на розподіленій програмній платформі (50 циклів виконання Cube), розгорнутій на одній фізичній машині. Основна мета цих експериментів була двояка. По-перше, ми прагнули проілюструвати здатність фреймворку Cube створювати та налаштовувати розподілені ланцюжки посередництва, які відповідають цілям архетипу. По-друге, ми мали на меті отримати кілька загальних показників продуктивності фреймворку Cube, щоб передбачити, як масштабуватиметься процес розв’язання залежно від кількості керованих платформ.

Експериментальна машина складалася з ноутбука (Core 2 Duo 3,06 ГГц і 4 Гб оперативної пам'яті). На цій машині розміщено платформу OSGi, яка містить пакет Cube Runtime і різні пакети розширення Cube. OSGi працює на JVM OpenJDK 1.6, що виконується в ОС Linux Ubuntu. Усі автономні менеджери Cube в експериментах створюються на цій машині. Кожен автономний менеджер виконується як окремий потік і підключається до інших автономних менеджерів через мережевий стек (через TCP/IP). Таким чином, автономні менеджери також можуть бути розгорнуті на різних фізичних машинах і мати однакову поведінку, тільки з більшими затримками.

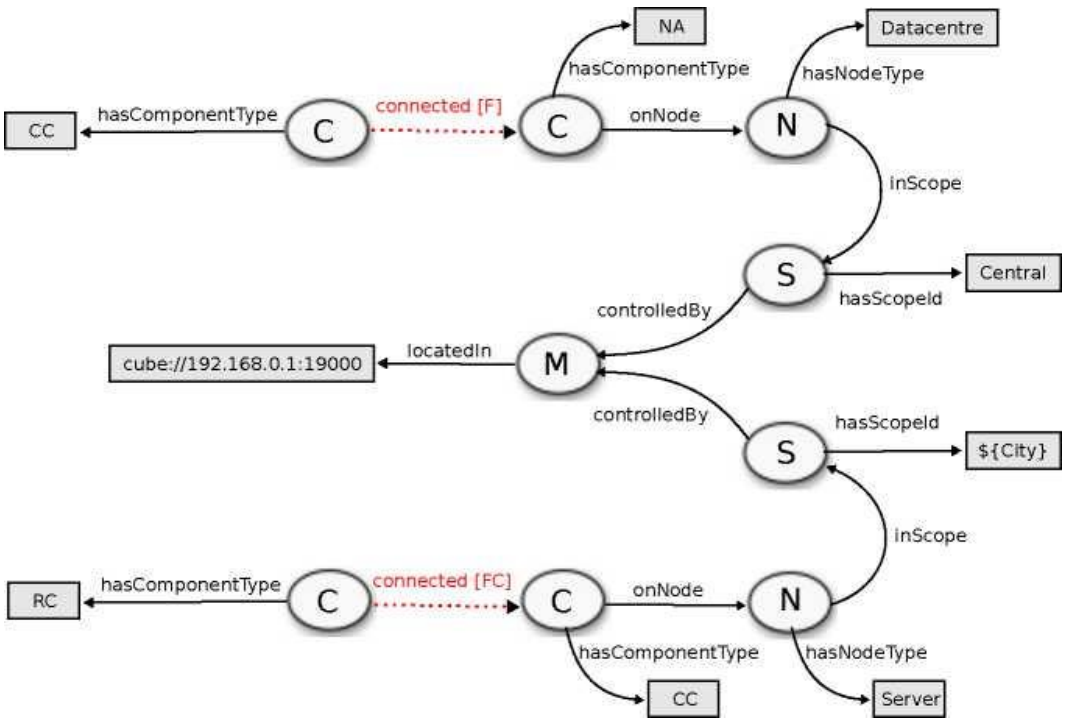


Рисунок 3.7 - Частина архетипу для самостійного створення та з’єднання компонентів

Ми оцінили реалізований прототип Cube за допомогою конфігурації, зображеної в таблиці 3.1.

Таблиця 3.1 - Автономні менеджери

AM n°	Host	Port	Role
1	localhost	19000	Master node
2	localhost	19001	National Datacentre
3-5	localhost	19100-19102	Grenoble Servers
6-10	localhost	19200-19204	Paris Servers
11-25	localhost	19300-19314	Grenoble Gateways
26-50	localhost	19500-19424	Paris Gateways

Загалом у нас є максимум п’ятдесят автономних менеджерів Cube. Ми провели серію експериментів, де для кожного експерименту ми збільшували кількість автономних менеджерів, що беруть участь – від одинадцять до п'ятдесяти. Метою було перевірити поведінку Cube, коли кількість автономних менеджерів (і, отже, керованих підсистем) збільшується.

Отже, у кожному окремому експерименті ми налаштовуємо розподілену платформу, щоб включати певну кількість керованих платформ (з одним AM на платформу); і ми спостерігаємо за кількістю керуючих повідомлень, якими обмінюються автономні менеджери, а також час вирішення, витрачений кожним автономним менеджером для забезпечення цілей Архетипу.

Автономні менеджери, пронумеровані від 1 до 10, присутні в усіх експериментах, оскільки вони керують вузлами, які належать інфраструктурі системи (сервери та центр обробки даних); ми припускаємо, що на цьому рівні немає варіацій. Навпаки, ми припускаємо, що система може включати різну кількість будинків, і ми моделюємо це, збільшуючи кількість з’єднаних будинків у кожному експерименті, як описано раніше.

Як зазначалося раніше, ми головним чином зацікавлені у перевірці здатності фреймворку Cube знаходити правильне рішення екземпляра для Архетипу; і в обчисленні деяких індикативних показників продуктивності, таких як кількість обмінених керуючих повідомлень і час вирішення окремих автономних менеджерів. Отже, для цілей цих експериментів ми не включили розширення вийок до автономних менеджерів; це просто продемонструвало б

здатність фреймворку Cube перетворювати рішення створення екземплярів автономних менеджерів, виражене через їхні відповідні моделі середовища виконання, у фактичні ланцюжки посередництва Cilia.

Однак найважливіше, як і в будь-якому децентралізованому управлінському рішенні, основні проблеми щодо масштабованості пов’язані з накладними витратами на зв’язок, спричиненими процедурами координації незалежних автономних менеджерів. Тому ми зосередилися на визначенні кількості повідомлень, якими обмінюються автономні менеджери під час процесу вирішення, а не на точних часових затримках, необхідних для цього процесу. Це дозволяє нам якісно оцінити масштабованість запропонованого децентралізованого рішення; тоді як точні характеристики продуктивності залежатимуть від продуктивності кожної платформи розподіленого розгортання.

Загалом було проведено 40 окремих експериментів, з одним додатковим шлюзом, включеним у кожен експеримент, відносно попереднього експерименту. Під час цих експериментів використовувалося максимум 40 шлюзів, з яких 15 домашніх шлюзів (з індексами AM від 11 до 25) і 25 домашніх шлюзів (з індексами AM від 26 до 50). Шлюзи додавались поступово протягом перших 15 експериментів; потім інші шлюзи були додані до решти 25 експериментів. У кожному експерименті автономні менеджери Cube повинні були починати процес вирішення з нуля, щоб створити екземпляр розподіленого ланцюга посередництва, який відповідав би цілям архетипу.

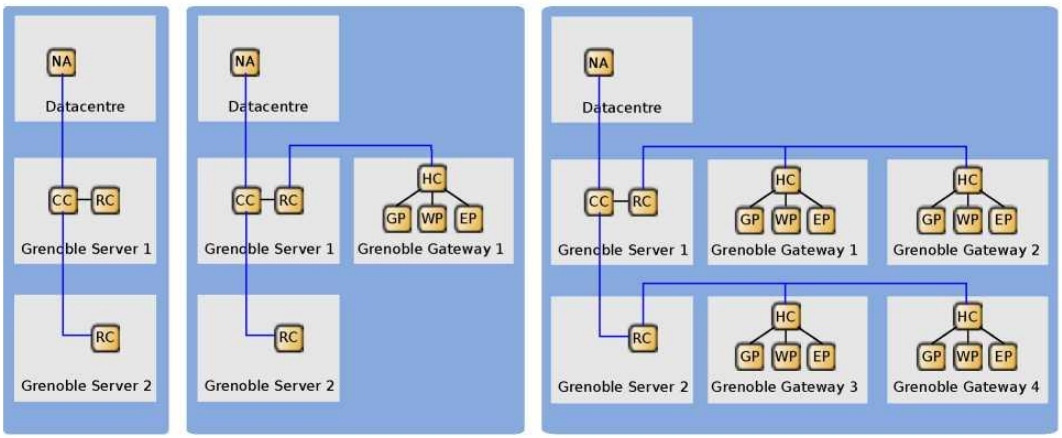


Рисунок 3.8 - Вигляд мережі для проведення тестування

Вирішується в трьох різних експериментальних умовах. Мета цих прикладів — показати, як фреймворку Cube вдається знаходити рішення для створення екземплярів, які відповідають архетипу, одночасно адаптуючись до кожної платформи розгортання. Це досягається за допомогою децентралізованого процесу вирішення архетипів, заснованого на співпраці автономних менеджерів (по одному для кожної платформи).

3.3 Застосування системи автономних обчислень в медичному проекті

Проведемо оцінку структури Cube у контексті медичного проекту. Медична допомога вдома стає все більш важливою та необхідною послугою в більшості розвинених країн . Дійсно, середнє населення цих країн прогресивно старіло протягом останніх десятиліть і, за оцінками, продовжить цю тенденцію протягом наступних десятиліть. У той же час, кількість людей похилого віку, які проживають вдома та/або знаходяться в ситуації залежності, зростає як через економічні фактори, так і через особистий вибір.

У цьому суспільному контексті важливо, щоб уряди та постачальники послуг покладалися на сучасні технології формування для створення належного середовища життя, пропонуючи інноваційні медичні послуги людям похилого віку. Технологічний прогрес у вбудованих пристроях, комунікаціях і цифрових домашніх послугах може бути використаний для забезпечення все більшої підтримки такого роду медичних послуг.

Однак запровадження таких послуг викликає численні технічні проблеми, пов’язані з впровадженням, підтримкою та розвитком таких послуг. В основному всі служби покладатимуться на дані, зібрані з конкретних пристроїв, розташованих у домівках, і передані через мережу до програм, пов’язаних із охороною здоров’я (рис. 3.9) . Розробка, розгортання та підтримка програмного забезпечення для передачі даних та інфраструктури для інтеграції пристроїв моніторингу з сервісними програмами є, у кращому випадку, складним завданням.

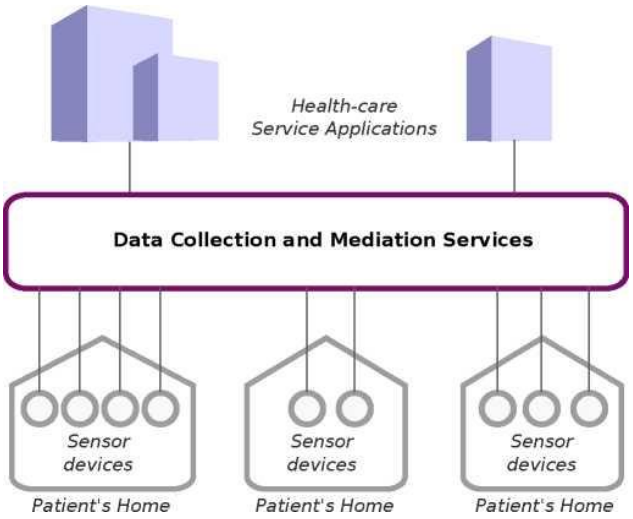


Рисунок 3.9 - Програмне забезпечення для обробки даних для додатків медичних послуг

Деякі з основних технічних проблем пов’язані з неоднорідністю пристроїв і даних, високою динамічністю пристроїв і поведінкою мешканців будинку. Крім того, користувачі можуть захотіти повторно використовувати свої пристрої під час підписки на різні послуги, які пропонують різні постачальники, що ще більше збільшує гетерогенність і загострює проблему. Користувачі також можуть змінювати свої вподобання та підписуватися на різних постачальників з часом, вимагаючи відповідної адаптації логіки посередництва. Нарешті, оскільки медичні послуги стають все більш популярними, базова інфраструктура та логіка посередництва повинні будуть відповідно масштабуватися та адаптуватися.

Ці міркування вказують на необхідність впровадження спеціального проміжного програмного забезпечення між домашніми пристроями та програмами охорони здоров’я. Такою системою посередництва має керувати сторонній постачальник з метою забезпечення чіткого технічного та адміністративного поділу між мешканцями будинку та постачальниками медичних послуг. Цей сторонній постачальник залучить певний досвід посередництва в даних, що дозволить іншим двом сторонам зосередитися на своїх конкретних проблемах.

На додаток до фактичної розробки такого проміжного ПЗ-посередника, його подальше розгортання та керування часом виконання викликають серйозніші проблеми. З боку користувачів, технологічні послуги догляду вдома не можуть управлятися жителями, які не є технічними експертами.

Наприклад, це включатиме операції розгортання, інстанціювання та конфігурація конкретних адаптерів пристроїв, які будуть вводити відстежувані дані через ланцюги посередництва, налаштовані для цільової служби. Подібним чином постачальникам послуг може бути складно та дорого вручну підключити свої послуги до ланцюгів посередництва, що потенційно потребує певних адаптацій. Нарешті, з боку провайдера посередництва головні проблеми полягають у адаптації інфраструктури посередництва до мінливого середовища та клієнтів, що розвиваються, а також у її виправленні у разі збоїв. Таким чином, медичний проект має на меті також надати пропоноване проміжне програмне забезпечення інтеграції можливостями автоматичного керування. Для вирішення цих проблем було прийнято структуру Cube.

Медичний проект передбачає два випадки використання:

- Моніторинг поведінки: спрямований на створення моделі, яка представляє поведінку пацієнтів у їхніх домівках, що допоможе виявити аномалії або відхилення від «нормальної» поведінки, що вказує на потенційні проблеми зі здоров’ям. Цей варіант використання вимагає оснащення будинку сенсорними пристроями, такими як датчики присутності.
- Моніторинг охорони здоров’я: має на меті полегшити доступ до даних, отриманих із медичних датчиків, як-от моніторів артеріального тиску чи рівня цукру, лікарями або родиною пацієнта, який перебуває під медичним наглядом. Ці дані збираються вдома та пересилаються на проміжні сервери для обробки, збереження та створення повної звітності.

Рішення, запропоноване медичним проектом, базується на таких інноваційних технологіях, як OSGi, iPOJO, OW2 Joram і фреймворк посередництва Cilia. Ми використали Cube для надання можливостей самоконтролю в контексті другого випадку використання «Моніторинг охорони

здоров’я». У наступному розділі ми детально описуємо цільову керовану систему, а далі ми описуємо вирішені проблеми та те, як Cube використовувався для забезпечення автономних можливостей.

Загальна система для сценарію моніторингу охорони здоров’я функціонує наступним чином. Фізіологічні дані збираються датчиками, встановленими вдома у пацієнта. Ці дані надсилаються одному або декільком проміжним сервер(и), де дані обробляються та зберігаються, щоб уповноважені спеціалісти, як-от медичний персонал або родина, могли отримати до них доступ і переглянути їх.

На домашньому рівні обробка отриманих даних не виконується. Натомість зібрані дані пересилаються безпосередньо до проміжного постачальника даних, де вони обробляються (наприклад, агрегуються, фільтруються або перекладаються), а потім надсилаються одному або кільком постачальникам персональних медичних карт (PHR), як-от Google Health , Microsoft Health Vault тощо. Медичні працівники та родина пацієнта можуть отримати доступ до оброблених даних, підписавшись і підключившись до таких PHR.

На рис. 3.10 показано ланцюжок посередництва, розроблений у рамках медичного проекту для сценарію моніторингу охорони здоров’я, для обробки та передачі даних між домашніми пристроями та постачальниками PHR. Ланцюжок посередництва досить простий, заснований на конвеєрному архітектурному стилі, що включає кілька посередників і зв’язків.

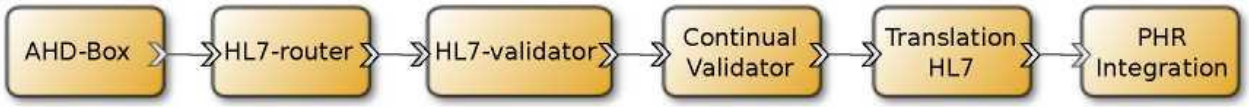


Рисунок 3.10 - Архітектура ланцюга посередництва для сценарію використання «Моніторингу охорони здоров’я»

Цей ланцюжок посередництва виконується на основі розподіленої платформи або інфраструктури, яка надається проміжним постачальником даних і розташована між будинком пацієнта та постачальником PHR. Точне визначення

та функціонування кожного посередника в наведеному ланцюжку виходить за межі нашого процесу перевірки. Оскільки ми зосереджені на початковому розгортанні та управлінні ланцюжком посередництва, нас цікавить лише архітектура системи посередництва та деякі її атрибути якості, як-от доступність, надійність тощо.

Точні параметри системи на домашньому рівні також виходять за межі наших інтересів самоуправління в цьому випадку використання. Загалом, на цьому рівні домашній шлюз підключається до кількох типів датчиків, таких як баланс, сфігмоманометр, частотомір і так далі. Цей шлюз підключено до Інтернету, щоб дозволити надсилати зібрані дані проміжному постачальнику посередництва. З нашої точки зору, необхідні драйвери датчиків і служби вже встановлені в шлюзі (наприклад, коли шлюз активований і підключений до Інтернету). Одним із важливих технічних аспектів, який слід звернути увагу на цьому рівні, є наявність локальної веб-служби (WS), яка дозволяє проміжному постачальнику посередництва отримувати зібрані дані через виклики веб-служби.

З архітектурної точки зору система посередництва, представлена в попередньому розділі, складається з відносно простого ланцюга посередництва. Однак, на додаток до функціональних можливостей, що стосуються конкретної програми, адміністратори хочуть забезпечити якість обслуговування системи (QoS), бажано за допомогою завдань адміністрування, які передбачають незначне втручання людини або взагалі без нього. Ми зображуємо ці цілі так:

1. Початкове розгортання, інстанціювання та конфігурація: автоматичне інстанціювання посередників, що утворюють ланцюжок посередництва, на серверах, наданих розподіленою інфраструктурою. Необхідні типи посередників і їх взаємозв'язки (рис. 3.10). Вхідні дані для ланцюга посередництва отримують зі шлюзів різних будинків-учасників. Кожен новий інтегрований будинок має бути підключений до цієї системи посередництва. Вихідні дані з ланцюжка посередництва надсилаються до списку партнерів-постачальників PHR.

2. Управління варіаціями робочого навантаження: автоматичне виконання балансування навантаження вхідних робочих навантажень на кількох серверах-посередниках. Це передбачає автоматичне дублювання ланцюга посередництва на кількох серверах, щоб підтримувати перевантаження та гарантувати, що час обробки вхідних повідомлень не перевищує попередньо визначений поріг. Подібним чином, коли вхідне навантаження зменшується, деякі сервери можуть бути зупинені (а їхні репліки ланцюга посередництва видалені) автоматично. Цей підхід може допомогти оптимізувати використання ресурсів.

3. Інтеграція нових постачальників PHR: автоматична зміна ланцюга посередництва (і всіх його реплік на серверах балансування навантаження), щоб оброблені дані могли надсилатися новим постачальникам PHR без зупинки та переналаштування всієї системи посередництва.

3.4 Представлення абстракції моделі та опис запропоноване рішення

У цьому підрозділі ми детально описуємо рішення, яке ми пропонуємо для вирішення проблем, розглянутих у попередньому підрозділі. На рис. 3.11 детально описана загальна архітектура запропонованої системи посередництва, розташованої в межах «Інфраструктури проміжного посередництва». Інфраструктура розділена на два основних кластери.

Перший кластер відіграє роль агрегації даних з різних будинків. Другий кластер виконує операції посередництва над отриманими даними. Як показано ланцюжок посередництва розподіляється між серверами в обох кластерах або для розділення логіки посередництва, або для цілей балансування навантаження. Ми використовуємо Cube для самостійного керування цією системою посередництва, а отже, для вирішення вищезгаданих проблем і вимог.

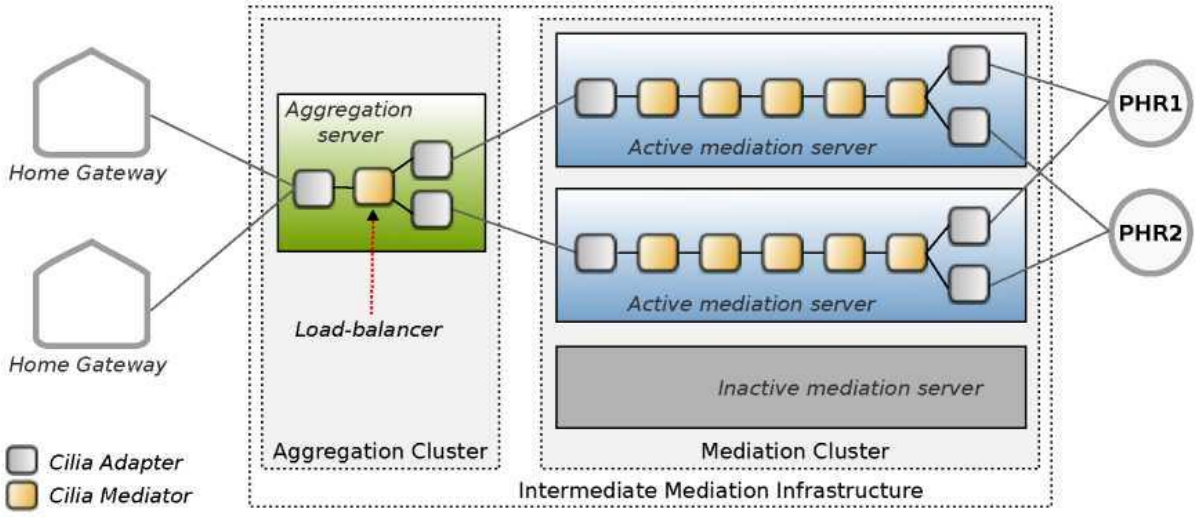


Рисунок 3.11 - Загальна архітектура посередництва з балансуванням навантаження між серверами-посередниками

Як зазначалося раніше, дані збираються з будинків-учасників за допомогою викликів веб-служби. Для цього ми використовуємо спеціальний адаптер Cilia Adapter. Цей адаптер вставляється на початку ланцюга посередництва та налаштований на зондування різних будинкових шлюзів для отримання локальних даних. Таким чином, цей адаптер розгортається та виконується на сервері в кластері агрегації.

Потім зібрані дані пересилаються на сервер у кластері посередництва (під назвою «Сервер посередництва»), який містить вищезгаданий ланцюжок посередництва. Для забезпечення розподіленого зв’язку між цими двома серверами потрібні ще два адаптери — один на сервері агрегатора (вихідний) і один на сервері-посереднику (вхідний). Як обговорювалося раніше, сервери в кластері-посереднику можна реплікувати (разом із ланцюжками посередництва), щоб забезпечити можливості балансування навантаження та уникнути системнихтемні перевантаження. Посередник балансування навантаження вводиться на сервер агрегації для розподілу зібраних даних між репліками сервера-посередника.

Дані, отримані в кінці ланцюга посередництва, надсилаються підключеним

постачальникам PHR. До кожного ланцюга посередництва (репліки) додається додатковий адаптер для перетворення оброблених даних у формат, специфічний для PHR. Нарешті, як і раніше, цей адаптер також використовується для надсилання даних до кожного віддаленого PHR за допомогою спеціальних протоколів і комунікаційних технологій.

Давайте тепер швидко подивимося на те, як розроблено балансування навантаження в поточному прототипі ; докладніше в цьому розділі. Ланцюг посередництва виконує однакові завдання для кожного отриманого повідомлення даних. Однак час обробки кожного повідомлення різний (наприклад, повідомлення мають різні розміри). Зазвичай адміністратори встановлюють угоди про рівень обслуговування, щоб гарантувати, що обробка повідомлень спричиняє обмежені затримки. Щоб забезпечити такі гарантії, щоразу, коли сервер завантажується на рівні, що перевищує фіксований рівень (наприклад, 70%), Cube має активувати інший сервер, дублювати ланцюжок посередництва та підключати його до балансувальника навантаження (розташованого на сервері агрегації), щоб отримати дані для обробки. Ця процедура зменшить кількість повідомлень для обробки перевантаженим сервером і гарантує, що затримка залишається в межах прийнятного інтервалу. І навпаки, коли кількість вхідних повідомлень зменшується та/або загальне навантаження на кластер може виконуватися меншою кількістю серверів, Cube автоматично вимикає додаткові сервери-посередники, щоб мінімізувати споживання ресурсів сервера.

Нижче ми пояснюємо різні етапи використання фреймворку Cube для керування типовою системою посередництва. Ми починаємо з окреслення необхідних абстракцій моделі. Пам’ятайте, що абстракції моделі використовуються для представлення елементів керованої системи під час. Потім ми окреслюємо список використовуваних розширень, деякі з яких були реалізовані спеціально для цього випадку використання. Нарешті, ми докладно описуємо запропоновану специфікацію Архетипу для опису різних цілей адміністрування.

Модельні абстракції

Ми використовуємо абстракції моделі, надані базовим розширенням фреймворку Cube. Зокрема, ми використовуємо абстрактні керовані елементи Component, Node, Scope та Master для написання специфікації Archetype. Таблиця 3.2 підсумовує відображення цих абстракцій на конкретні елементи системи, якими керують у цьому випадку використання. Він показує, як базові абстракції Cube розширені для представлення предметно-спеціальних абстракцій для моделювання цього конкретного випадку використання. Наприклад, концепція компонента відображається на медіаторах і адаптерах Cilia за допомогою розширення Cilia. Було реалізовано додаткові розширення для інтеграції PHR і балансування навантаження.

Таблиця 3.2 - Абстракції моделі для другого прикладу

Model Element	Semantic
Component	Used as an abstraction for Cilia Mediators and Adapters (like in the first case study).
Node	Used as an abstraction for OSGi platforms. Each OSGi platform contains a local mediation chain with a set of Mediators (and Adapters). It has the following two properties: • **cpu**: contains the medium value of the monitored CPU load, over a fixed interval. It is set by the Load Balancing (LB) Extension (discussed below). • **active**: if set to "true", the corresponding Node is considered to be active. This means that the managed server modelled by this Node abstraction is currently being employed in the mediation infrastructure (to host and execute mediation chains).
Scope	Used as a grouping abstraction for a set of OSGi platforms to simplify goal descriptions and to help coordinate the distributed resolution process. We use this model abstraction to regroup the OSGi platforms of the same cluster. We need to set the following property of the Scope element: • **scopeId**: the unique identifier of the Scope element. Since we only have two scopes in our case study - one for regrouping the nodes of the mediation cluster and the other for the nodes of the aggregation cluster - we do not use scope types in this use case; the identifier of each scope will suffice.
Master	Used as a global leader abstraction for coordinating the system's Scopes. In this case study, we instantiate the Master element in a dedicated Cube Autonomic Manager. It only holds the two references to the leaders of the two Scopes.

Таблиця 3.3 підсумовує список розширень, використаних для цього прикладу. Серед них два нових розширення були реалізовані спеціально для виконання конкретних вимог цього другого прикладу. Ці нові розширення керують механізмами балансування навантаження та інтеграцією нових постачальників PHR.

Таблиця 3.3 - Використовувані розширення та їх властивості конфігурації

Extension Name	Extension Namespace	Configuration Properties
Core Extension	fr.liglab.adele.cube.core	master
Cilia Extension	fr.liglab.adele.cube.cilia	jms.server
		jms.port
PHR Extension	fr.liglab.adele.cube.phr	interval
		phr-providers-file
LB Extension	fr.liglab.adele.cube.loadbalancing	interval
		max-limit

Основне розширення надає всі абстракції моделі, що використовуються в цьому варіанті використання – компонент, вузол, область і майстер – а також більшість пов’язаних з ними властивостей архетипу (властивості опису та цілі), а також пов’язані з ними резолвери.

Основна роль розширення Cilia полягає в синхронізації стану Runtime Model у Cube Autonomic Manager із фактичним ланцюгом Cilia в керованій. Давайте проілюструємо це на прикладі випадку, коли Cube вирішує видалити (частину) ланцюжка посередництва. Коли розпізнавач архетипів Cube видаляє один або кілька екземплярів Компонента з моделі виконання (що представляють медіатори Cilia), розширення Cilia відповідно видаляє відповідні медіатори Cilia з конкретного ланцюга посередництва. Щоб забезпечити узгодженість і уникнути втрати даних, кожен посередник Cilia видаляється лише після обробки всіх його буферизованих даних, які очікують на обробку, і завершення всіх поточних операцій обробки.

Ми запропонували та впровадили розширення PHR, щоб дозволити інтегрувати нових постачальників PHR у систему медіації. Це розширення читає файл, розташований у локальній файловій системі або на віддаленому сервері (вказано у властивості конфігурації phr-providers-file). Цей файл містить список

усіх провайдерів PHR, розпізнаних системою. Розширення PHR зчитує цей файл у заздалегідь визначений час (зазначений у властивості конфігурації інтервалу), щоб визначити, чи приєднався або вийшов із системи будь-який новий постачальник (відповідно доданий або видалений зі списку PHR). Для кожного запису розширення PHR створює новий екземпляр компонента типу «PHR-інтеграція», що має властивість «adapter-component», що містить ім’я адаптера Cilia, відповідального за надсилання даних конкретному постачальнику PHR. Резолвер Archetype інтегрує цей екземпляр у кінці ланцюга посередництва (як зазначено в Archetype).

Розширення балансування навантаження (під назвою LB Extension) дозволяє активувати або дезактивувати вузли залежно від загального навантаження на активні сервери-посередники в будь-який момент часу. Це розширення може змінювати значення властивості «active» екземпляра Node. Ми використовуємо цю «активну» властивість в Архетипі, щоб вказати, що робити у випадку зміни цієї властивості. Коли екземпляр Node активний (для властивості «active» встановлено значення «true»), ланцюжок посередництва, специфічний для випадку використання, створюється та налаштовується для отримання даних від посередника Load Balancing Cilia (розташованого на сервері агрегації). Коли вузол стає неактивним (для властивості «active» встановлено значення «false»), ланцюжок посередництва припиняється та видаляється з відповідного сервера-посередника.

Розширення LB виконує наступні дві ролі:

1. Він відстежує споживання процесора виконуваними серверами та оновлює властивість «cpu» відповідного екземпляра Node у частині Runtime Model Cube (рис. 3.12) . У реальному реалізованому розширенні LB ми просто надаємо змодельовані значення процесора випадковим чином.

2. Він обчислює глобальне навантаження всіх активних вузлів у кластері-посереднику та вирішує активувати або деактивувати деякі вузли, щоб залишатися в межах прийнятного інтервалу навантаження. Ця функція

викликається з кожним інтервалом (зазначеним як властивість конфігурації розширення LB).

З огляду на наші цілі оцінки, основна роль змодельованих значень ЦП і алгоритму оцінки навантаження полягає в тому, щоб запустити можливості Cube для автоматичної реплікації або видалення ланцюжків посередництва, таким чином підкреслюючи функції балансування навантаження та масштабованості системи Cube. Більш реалістичні функції моніторингу, оцінки навантаження та забезпечення сервера можуть бути запроваджені в реальних сценаріях використання.

На рис. 3.12 показаний повний приклад розширень, які використовуються для цього випадку використання. На цьому Рис. ми не показуємо розширення Core Extension, яке є в усіх автономних менеджерах.

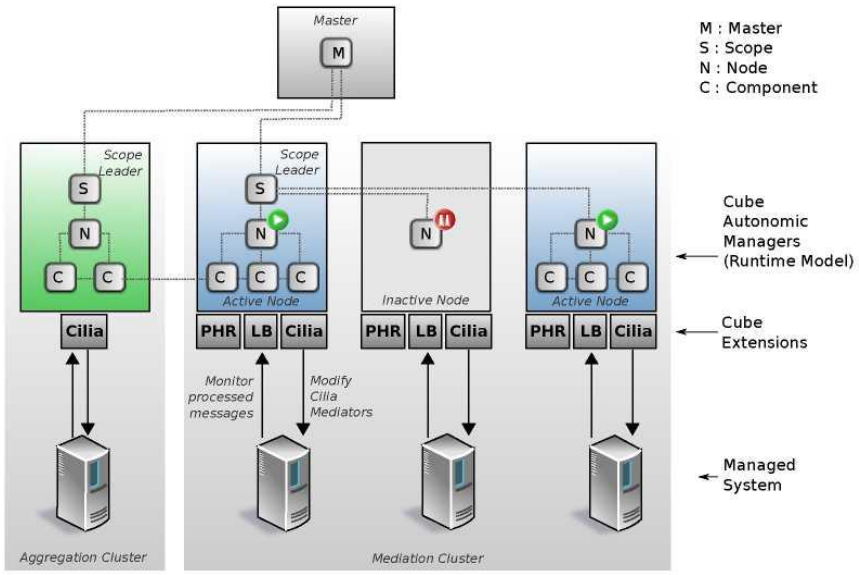


Рисунок 3.12 - Розширення куба

Менеджер використовується для виконання ролі головного вузла, що містить список лідерів області. Зміст моделі виконання автономних менеджерів відповідає тому, що визначено як цілі в Архетипі (докладніше далі). На серверах агрегації ми використовуємо лише розширення Cilia (і розширення Core) для керування локальним ланцюгом посередництва. Однак на рівні кластера

посередництва ми додаємо розширення PHR і LB до кожного автономного менеджера, щоб забезпечити функцію балансування навантаження та динамічну інтеграцію нових постачальників PHR.

На рисунку 3.13 (a, b, c) показано різні стани чотирьох серверів у кластері-посереднику в різний час.

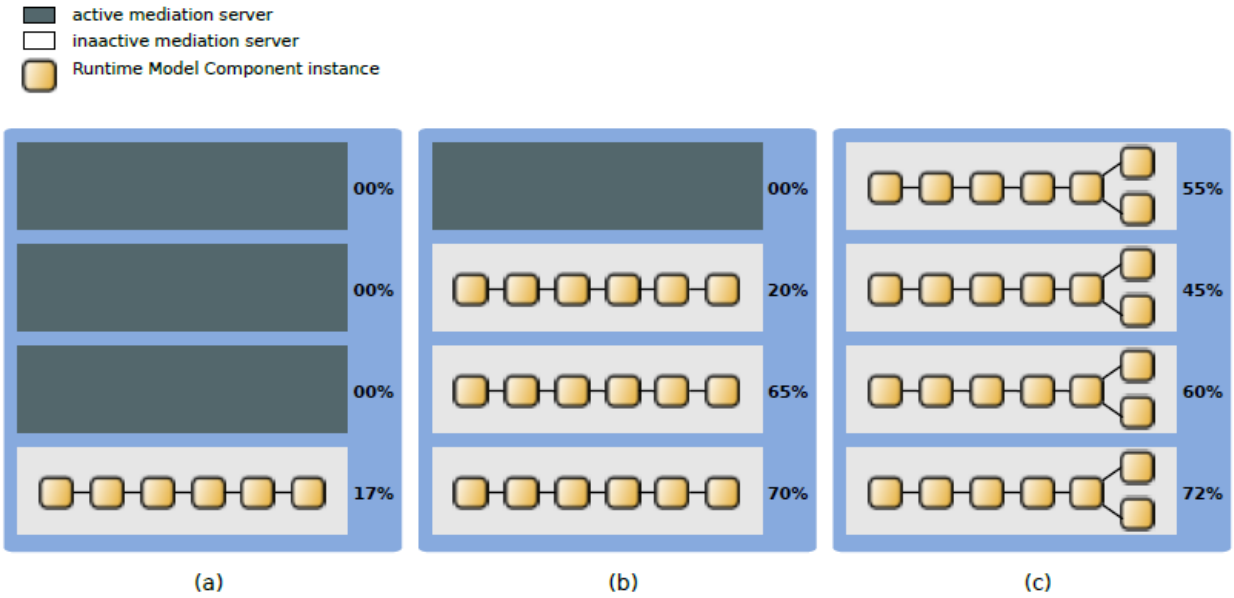


Рисунок 3.13 - Балансування навантаження на основі середнього навантаження ЦП

Для кожного стану сірі прямокутники позначають неактивний сервер-посередник, тоді як білі прямокутники позначають активний сервер посередництва. У першому стані (a) активовано єдиний сервер-посередник («medsrv1»).

Його навантаження на процесор (17%) не перевищує попередньо встановленого порогу (70%). У другому стані (b) сервер навантаження ЦП перевищило цей ліміт, що призвело до активації Cube іншого сервера-посередника («medsrv2»). Оскільки активації цього нового сервера-посередника було недостатньо, щоб знизити середнє навантаження на сервер фіксований поріг, Cube активував третій сервер-посередник (“medsrv3”). У цьому новому стані серед навантаження на ЦП на кожному з трьох активних серверів не

перевищує визначений поріг (70%). В останньому у випадку (c) усі сервери-посередники були активовані для підтримки накладних витрат.

Якщо загальне навантаження все ще перевищує фіксований ліміт і якщо немає додаткових серверів-посередників для активації, Cube покаже відповідне адміністративне повідомлення на консолі, вказуючи на проблему та вказуючи системним адміністраторам виправити ситуацію. У реальній виробничій системі може бути Cube запрограмований на надсилання електронного листа безпосередньо адміністраторам, щоб повідомити їм про поточну ситуацію.

Висновки до розділу 3

Отже, загальний формат Runtime Model і протокол зв’язку Cube дозволяють керувати різнорідними частинами системи, взаємодіяти для досягнення глобального рішення для керованої системи. Багаторазове використання стосується придатності компонентів і підсистем для різних програм або випадків використання. Повторне використання допомагає запобігти дублюванню компонентів і, отже, може скоротити час впровадження нових програм.

ВИСНОВКИ

В кваліфікаційній роботі виконано дослідження та проведено порівняльний аналіз моделей, методів та алгоритмів побудови децентралізованого фреймворку на прикладі використання систем автономних обчислень. Було розглянуто процес імплементації моделей автономних обчислень на основі фреймворку з децентралізованою архітектурою. Проведене дослідження підтвердило актуальність та важливість використання децентралізованих моделей у сучасних інформаційних системах. В результаті виконаного дослідження з порівняльного аналізу моделей та методів побудови децентралізованої архітектури можна зробити декілька важливих висновків.

У роботі було виконано оцінку структури запропонованого фреймворку Cube у двох варіантах. В першому варіанті використання - моніторинг споживання внутрішні ресурси - ми провели кількісну оцінку шляхом вимірювання операцій самоуправління реалізовано платформою Cube для забезпечення самостійного створення програми-посередника розподілених даних. Показано, що виконання такої складної операції управління може вимагати лише обмежений час, у порівнянні з ручним або спеціальним створенням і налаштуванням процесів посередництва Cilia. У другому варіанті використання – системі моніторингу охорони здоров’я – ми провели якісний аналіз та оцінку системи автономних обчислень. Результати представленого дослідження можуть бути використані для вибору оптимальних стратегій та інструментів децентралізованих архітектур та систем автономних обчислень у різних сценаріях та завданнях і можуть бути основою для подальших досліджень та розробки нових підходів у сфері обробки великих об’ємів даних. Узагальнюючи, можна стверджувати, що децентралізовані архітектури та автономні обчислення мають значний потенціал для підвищення ефективності, надійності та безпеки сучасних інформаційних систем. Їхнє впровадження сприяє розвитку нових технологій і бізнес-моделей, що відповідають вимогам швидко змінюваного цифрового світу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. F. Akkawi, K. Akkawi, A. Bader, M. Ayyash, D. Fletcher, and K. Alzoubi, Software adaptation: A conscious design for oblivious programmers, Aerospace Conference, 2007 IEEE, march 2007, pp.

2. H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T.F. Knight Jr, R. Nagpal, E. Rauch, G.J. Sussman, and R. Weiss, Amorphous computing, Communications of the ACM 43 (2000), no. 5, 74–82.

3. Ip Agarwala, Yuan Chen, Dejan Milojicic, and Karsten Schwan, Qmon: Qos- and utility-aware monitoring in enterprise systems, In The 3rd IEEE International Conference on Autonomic Computing, IEEE Computer Society, 2006, pp. 124–133.

4. Robert Allen, Remi Douence, and David Garlan, Specifying and analyzing dynamic software architectures, Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98) (Lisbon, Portugal), March 1998, An expanded version of a the paper "Specifying Dynamism in Software Architectures," which appeared in the Proceedings of the Workshop on Foundations of Component-Based Software Engineering, September 1997.

5. Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney, A survey of adaptive optimization in virtual machines, PROCEEDINGS OF THE IEEE, 93(2), 2005. SPECIAL ISSUE ON PROGRAM GENERATION, OPTIMIZATION, AND ADAPTATION, 2004.

6. Nejla Amara-Hachmi, A framework for building adaptive mobile agents, Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems (New York, NY, USA), AAMAS '05, ACM, 2005, pp. 1369–1369.

7. OSGi Alliance, Osgi service platform core specification, release 4.1, http://www.osgi.org/Specifications, 2007.

8. R.J. Anthony, Emergence: a paradigm for robust and scalable distributed applications, Autonomic Computing, 2004. Proceedings. International Conference on, may 2004, pp. 132 – 139.

9. Richard Anthony, Mariusz Pelc, and Haffiz Shuaib, The interoperability challenge for autonomic computing, EMERGING 2011, The Third International Conference on Emerging Network Intelligence, 2011, pp. 13–19.

10. Karl Johan Astrom, Challenges in control education, Proc. of the 7th IFAC Symposium on Advances in Control Education, Plenary Lecture, June, 2007, pp. 21–23.

11. G. Blair, N. Bencomo, and R.B. France, Models@ run.time, Computer 42 (2009), no. 10, 22 –27.

12. Sara Bouchenak, Fabienne Boyer, Daniel Hagimont, Sacha Krakowiak, Adrian Mos, Noel De Palma, Vivien Quema, and Jean bernard Stefani, Architecture-based autonomous repair management: An application to j2ee clusters, In 24th IEEE Symposium on Reliable Distributed Systems (SRDS-2005, 2005, pp. 13–24.

13. Francois Bousquet, Innocent Bakam, Hubert Proton, and Christophe Le Page, Cormas: common-pool resources and multi-agent systems, Tasks and Methods in Applied Artificial Intelligence (1998), 826–837.

14. J.S. Bradbury, J.R. Cordy, J. Dingel, and M. Wermelinger, A survey of selfmanagement in dynamic software architecture specifications, Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, ACM, 2004, p. 33.

15. L. Bass, P. Clements, and R. Kazman, Software architecture in practice, Addison-Wesley Professional, 2003.

16. Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani, The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems, Softw. Pract. Exper. 36 (2006), no. 11- 12, 1257–1284.

17. Gordon Blair, Thierry Coupaye, and Jean-Bernard Stefani, Component-based architecture: the fractal initiative, Annals of Telecommunications 64 (2009), no. 1, 1–4. 3

18. Johann Bourcier, Ada Diaconescu, Philippe Lalanda, and Julie A. McCann, Autohome: An autonomic management framework for pervasive home applications, ACM Trans. Auton. Adapt. Syst. 6 (2011)

19. Jacob Beal, Functional blueprints: An approach to modularity in grown systems, Swarm Intelligence 5 (2011), no. 3, 257–281.

20. Fran Berman, Geoffrey Fox, and Anthony JG Hey, Grid computing: making the global infrastructure a reality, vol. 2, Wiley, 2003.

21. Jean Bezivin and Olivier Gerbe, Towards a precise definition of the omg/mda framework, Proceedings of the 16th IEEE international conference on Automated software engineering (Washington, DC, USA), ASE ’01, IEEE Computer Society, 2001,

22. Rafael H Bordini, Jomi Fred Hübner, and Michael Wooldridge, Programming multiagent systems in agentspeak using jason, vol. 8, Wiley-Interscience, 2007.

23. Ozalp Babaoglu, Mark Jelasity, and Alberto Montresor, Gossip-based selfmanaging services for large scale dynamic networks, Service Management and Self-Organization in IP-based Networks (Dagstuhl, Germany) (Matthias Bossardt, Georg Carle, D. Hutchison, Hermann de Meer, and Bernhard Plattner, eds.), Dagstuhl Seminar Proceedings, no. 04411, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

24. Ozalp Babaoglu, Márk Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, Aad van Moorsel, and Maarten van Steen, Self-star properties in complex information systems: Conceptual and practical foundations (lecture notes in computer science), Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

25. Huhns, Agents and service-oriented computing for autonomic computing: A research agenda, IEEE Internet Computing 13 (2009), 82–87.

26. Bill Burke and Richard Monson-Haefel, Enterprise javabeans 3.0, 5. ed., O’Reilly, Beijing, 2006.

27. Johann Bourcier, Auto-home: une plate-forme pour la gestion autonomique d’applications pervasives, Ph.D. thesis, University Joseph Fourrier, 2008.

28. Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos, Tropos: An agent-oriented software development methodology, Autonomous Agents and Multi-Agent Systems 8 (2004), no. 3, 203–236.

29. S. Bouchenak, N. De Palma, D. Hagimont, and C. Taton, Autonomic management of clustered applications, Cluster Computing, IEEE International Conference on 0 (2006), 1–11.

30. Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf, Jadex: A short overview, Main Conference Net. ObjectDays, vol. 2004, 2004, pp. 195–207.

31. Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa, Jade: a fipa2000 compliant agent development environment, Proceedings of the fifth international conference on Autonomous agents, ACM, 2001, pp. 216–217.

32. J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills, and Y. Diao, Able: a toolkit for building multiagent autonomic systems, IBM Syst. J. 41 (2002), no. 3, 350–371.

33. David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés, Automated reasoning on feature models, Proceedings of the 17th international conference on Advanced Information Systems Engineering (Berlin, Heidelberg), CAiSE'05, Springer-Verlag, 2005, pp. 491–503.

34. Mike Burrows, The chubby lock service for loosely-coupled distributed systems, Proceedings of the 7th symposium on Operating systems design and implementation (Berkeley, CA, USA), OSDI '06, USENIX Association, 2006, pp. 335–350.

35. Bogdan Alexandru Caprarescu, Robustness and scalability: a dual challenge for autonomic architectures, Proceedings of the Fourth European Conference on Software Architecture: Companion Volume (New York, NY, USA), ECSA '10, ACM, 2010, pp. 22–26.

36. Krzysztof Czarnecki and Ulrich W. Eisenecker, Generative programming: methods, tools, and applications, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

37. Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano, Autonomic computing through reuse of variability models at runtime: The case of smart homes, Computer 42 (2009), no. 10, 37–43.

38. James Coplien, Daniel Hoffman, and David Weiss, Commonality and variability in software engineering, IEEE Softw. 15 (1998)

39. Alain Cardon and Mhamed Itmi, Multi agent modeling of self adaptive system for large scale complex systems, Proceedings of the 2009 Summer Computer Simulation Conference (Vista, CA), SCSC ’09, Society for Modeling & Simulation International, 2009, pp. 476–481.

40. Paul C. Clements and Linda M. Northrop, Software architecture: An executive overview, 1996.

41. Jaron C Collis, Divine T Ndumu, Hyacinth S Nwana, and Lyndon C Lee, The zeus agent building tool-kit, BT Technology Journal 16 (1998), no. 3, 60–68.

ДОДАТКИ

Додаток А

На рисунку А.1 показано діаграму класів UML інтерфейсу служби адміністрування. Цей інтерфейс реалізовано класом Java CubeRuntime.

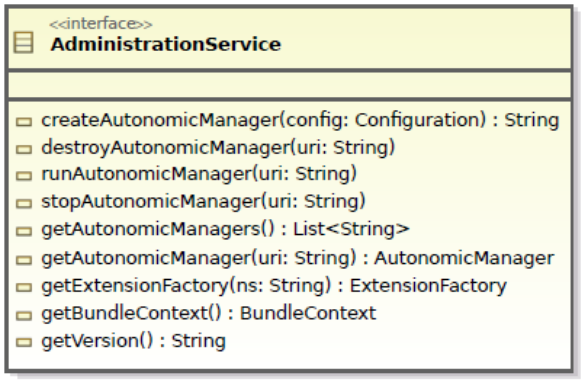


Рисунок А.1 - Administration Service Java Interface

У наступному лістингу показано деталі реалізації змодельованого елемента «Componentt».

Лістинг А.1. Клас Java змодельованого елемента "Componentt".

```
1  public class Component extends ManagedElement {
2
3      public static final String NAME = "Component";
4      public static final String CORE_COMPONENT_ID = "id";
5      public static final String CORE_COMPONENT_TYPE = "type";
6      public static final String CORE_COMPONENT_NODE = "node";
7      public static final String CORE_COMPONENT_INPUTS = "inputs";
8      public static final String CORE_COMPONENT_OUTPUTS = "outputs";
9
10     public Component(String amUri) {
11         super(amUri);
12     }
13
14     public Component(String amUri, Properties properties) throws
           PropertyExistException, InvalidNameException {
15         super(amUri, properties);
16         setNamespace(CoreExtensionFactory.NAMESPACE);
17         setName(NAME);
18     }
19
20     public void setComponentId(String component_identifier) {
```

```
21          try {
22              if (this.getAttribute(CORE_COMPONENT_ID) == null)
23                  this.addAttribute(CORE_COMPONENT_ID, component_identifier);
24              else
25                  this.updateAttribute(CORE_COMPONENT_ID, component_identifier);
26          } catch (PropertyNotExistException e) {
27              e.printStackTrace();
28          } catch (InvalidNameException e) {
29              e.printStackTrace();
30          } catch (PropertyExistException e) {
31              e.printStackTrace();
32          }
33      }
34
35      public String getComponentId() {
36          return this.getAttribute(CORE_COMPONENT_ID);
37      }
38
39      public void setComponentType(String component_type) {
40          try {
41              if (this.getAttribute(CORE_COMPONENT_TYPE) == null)
42                  this.addAttribute(CORE_COMPONENT_TYPE, component_type);
43              else
44                  this.updateAttribute(CORE_COMPONENT_TYPE, component_type);
45          } catch (PropertyNotExistException e) {
46              e.printStackTrace();
47          } catch (InvalidNameException e) {
48              e.printStackTrace();
49          } catch (PropertyExistException e) {
50              e.printStackTrace();
51          }
52      }
53
54      public String getComponentType() {
55          return this.getAttribute(CORE_COMPONENT_TYPE);
56      }
57
58      public void setNode(String node_url) {
59          Reference r = null;
60          try {
61              r = this.addReference(CORE_COMPONENT_NODE, true);
62          } catch (InvalidNameException e) {
63              e.printStackTrace();
64          }
65          r.addReferencedElement(node_url);
66      }
67
68      public String getNode() {
69          Reference r = getReference(CORE_COMPONENT_NODE);
70          if (r != null && r.getReferencedElements().size() > 0) {
71              return r.getReferencedElements().get(0);
72          }
73          return null;
74      }
75
76      public boolean addInputComponent(String compURI) {
77          Reference r = null;
```

```
78          try {
79              r = addReference(CORE_COMPONENT_INPUTS, false);
80          } catch (InvalidNameException e) {
81              e.printStackTrace();
82          }
83          return r.addReferencedElement(compURI);
84      }
85
86      public List<String> getInputComponents() {
87          Reference r = this.getReference(CORE_COMPONENT_INPUTS);
88          if (r != null) {
89              return r.getReferencedElements();
90          }
91          return new ArrayList<String>();
92      }
93
94      public boolean addOutputComponent(String compURI) {
95          Reference r = null;
96          try {
97              r = addReference(CORE_COMPONENT_OUTPUTS, false);
98          } catch (InvalidNameException e) {
99              e.printStackTrace();
100         }
101         return r.addReferencedElement(compURI);
102     }
103
104     public List<String> getOutputComponents() {
105         Reference r = this.getReference(CORE_COMPONENT_OUTPUTS);
106         if (r != null) {
107             return r.getReferencedElements();
108         }
109         return new ArrayList<String>();
110     }
111 }
```

метадані

Заголовок

Імплементація моделей автономних обчислень на основі фреймворку з децентралізованою архітектурою

Автор

Луканюк Л.В. Науковий керівник / Експерт

підрозділ

King Danylo University

Тривога

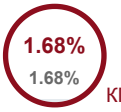
У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв	ß	0
Інтервали	A→	0
Мікропробіли	·	0
Білі знаки	ß	0
Парафрази (SmartMarks)	a	11

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

КП 1



КП 2

25
Довжина фрази для коефіцієнта подібності 2

8136
Кількість слів

65149
Кількість символів

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз — Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)	
1	http://repository.ukd.edu.ua/bitstream/handle/123456789/388/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0%20%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0%20%D0%9B%D0%B8%D1%82%D0%B2%D0%B0%D0%BA.pdf?sequence=1	68	0.84 %
2	http://repository.ukd.edu.ua/bitstream/handle/123456789/390/%D0%9C%D0%B0%D0%BD%D1%82%D1%83%D0%BB%D1%8F%D0%BA%20%D0%94.%D0%92.%20%D0%9A%D0%A0.pdf?sequence=1	36	0.44 %
3	http://repository.ukd.edu.ua/bitstream/handle/123456789/390/%D0%9C%D0%B0%D0%BD%D1%82%D1%83%D0%BB%D1%8F%D0%BA%20%D0%94.%D0%92.%20%D0%9A%D0%A0.pdf?sequence=1	33	0.41 %