КВАЛІФІКАЦІЙНА РОБОТА

Група МІПЗс-22

Куцела Я.Т.

2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

на правах рукопису

Куцела Ярослав Тарасович

УДК 004.4

Імплементація моделей покращення керованості великих масштабованих розподілених обчислювальних систем

Спеціальність 121 – «Інженерія програмного забезпечення»

Кваліфікаційна робота на здобуття кваліфікації магістра

Нормоконтроль

_____Стисло О.В.

(підпис, дата, розшифрування підпису)

Студент

_____ Куцела Я.Т.

(підпис, дата, розшифрування підпису)

Допускається до захисту

Завідувач кафедри

_____к.т.н., доц. Ващишак С.П.

(підпис, дата, розшифрування підпису)

Керівник роботи

_____ к.т.н., доц. Демчина М.М.

(підпис, дата, розшифрування підпису)

Івано-Франківськ – 2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

Освітній ступінь: «магістр»

Спеціальність: 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____

« 19 » лютого 2024 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Куцили Ярослава Тарасовича

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи

Імплементація моделей покращення керованості великих масштабованих розподілених обчислювальних систем

керівник роботи:

Демчина Микола Миколайович, кандидат технічних наук, доцент

затверджена наказом вищого навчального закладу від « 26 » червня 2023 року № 32/1 с

2. Термін подання студентом роботи 16.02.2024

3. Вихідні дані роботи: Формальні моделі, методи та алгоритми.

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

1. Аналіз методологій керування обчислювальними системами.
2. Структура моделі адаптивного керування системою.
3. Побудова моделі та шаблонів взаємодії обчислювального середовища
4. Імплементація концепції самокерованих додатків

5. Дата видачі завдання 29.06.2023

КОНСУЛЬТАНТИ РОЗДІЛІВ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Розділ	Консультант (прізвище, ініціали та посада)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1.	Огляд та аналіз методологій керування розподіленими обчислювальними системами	26.09.2023	Виконано
2.	Дослідження структури моделі самоналаштовуваного адаптивного керування системою	20.10.2023	Виконано
3.	Побудова моделі та шаблонів взаємодії розподіленого обчислювального середовища	15.11.2023	Виконано
4.	Імплементація концепції самокерованих додатків	30.11.2023	Виконано
5.	Формування висновків	09.12.2023	Виконано
6.	Оформлення пояснювальної записки	22.12.2023	Виконано
7.	Оформлення графічного матеріалу та підготовка до захисту роботи	11.01.2024	Виконано

Студент _____ Куцела Я.Т.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Демчина М.М.
(підпис) (прізвище та ініціали)

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

Сторінка	Опис графічного матеріалу	Сторінка	Опис графічного матеріалу
18	Проста автономна обчислювальна архітектура	58	Композитний фрактальний компонент HelloWorld із двома підкомпонентами клієнт і сервер
22	Фрактальна компонентна модель	60	Ієрархія елементів управління
24	Децентралізовані неструктуровані однорангові мережі	65	Додаток в якому клієнтський компонент підключено до групи з двох компонентів служби

24	Гібридні неструктуровані однорангові мережі	69	Події та дії в циклі самовідновлення програми
30	Багаторівнева програма Web 2.0 з контролером еластичності, розгорнута в хмарному середовищі	76	Шаблони взаємодії а) ефект стигметрії б) пряма взаємодія
33	Традиційна та самокерована трирівнева архітектура	77	Шаблони взаємодії а) ієрархічне управління б) спільні елементи керування
40	Абстрактний (ліворуч) і конкретний (праворуч) вигляд конфігурації. Коробки представляють вузли або віртуальні машини, кола представляють компоненти	80	Функціональний дизайн YASS
45	Абстрактна конфігурація самокерованої програми	81	Цикл керування самовідновленням для відновлення реплік файлів
48	Архітектура пропонованої системи	82	Контур керування самоналаштуванням для додавання сховища
52	Етапи виклику методу в системі		

АНОТАЦІЯ

Кваліфікаційна робота присвячена процесам імплементації моделей покращення керованості великих масштабованих розподілених обчислювальних систем шляхом впровадження архітектури автономних обчислень у повністю децентралізований спосіб, щоб відповідати вимогам великомасштабних розподілених систем.

В першому розділі здійснено аналіз методологій керованості великими розподіленими обчислювальними системами. Описано концепції розподілених обчислювальних систем, сутність автономних обчислень. Досліджено фрактальну компонентну модель, проведено огляд сучасних мережевих сервісів та процесів хмарних обчислень та еластичних послуг.

В другому розділі наведена структура моделі самоналаштовуваного адаптивного керування розподіленою обчислювальною системою, описано сучасний стан справ в області самоуправління великих розподілених систем, досліджено підхід до побудови розподіленої системи керування компонентами. Здійснено розробку структури системи самоналаштовуваного адаптивного керування обчислювальною системою і наведена архітектура пропонованої системи керування.

В третьому розділі описано процес побудови моделі та шаблонів взаємодії розподіленого масштабованого обчислювального середовища, представлено алгоритм розробки самоналаштовуваних додатків. Описана концепція адаптивного програмування та виконано програмування функціональних компонентів і груп компонентів розподіленої системи. Проведена методологія проектування для самоуправління в розподілених обчислювальних середовищах.

КЛЮЧОВІ СЛОВА: ХМАРНІ ОБЧИСЛЕННЯ, РОЗПОДІЛЕНА СИСТЕМА, ФРАКТАЛЬНА МОДЕЛЬ, ФУНКЦІОНАЛЬНІ КОМПОНЕНТИ, ВЛАСТИВІСТЬ САМОНАЛАШТОВУВАНОСТІ.

SUMMARY

The qualification work is devoted to the processes of implementing models for improving the controllability of large-scale distributed computing systems by implementing the architecture of autonomous computing in a fully decentralized way to meet the requirements of large-scale distributed systems.

In the first section, an analysis of controllability methodologies for large distributed computing systems is carried out. The concepts of distributed computing systems, the essence of autonomous computing are described. The fractal component model was studied, an overview of modern network services and processes of cloud computing and elastic services was conducted.

In the second chapter, the structure of the model of self-configuring adaptive control of a distributed computing system is presented, the current state of affairs in the field of self-management of large distributed systems is described, and the approach to building a distributed component control system is investigated. The structure of the system of self-adjusting adaptive control of the computer system was developed and the architecture of the proposed control system was presented.

The third chapter describes the process of building a model and patterns of interaction of a distributed scalable computing environment, presents an algorithm for developing self-configuring applications. The concept of adaptive programming is described and the programming of functional components and groups of components of a distributed system is performed. The design methodology for self-management in distributed computing environments is carried out.

KEY WORDS: CLOUD COMPUTING, DISTRIBUTED SYSTEM, FRACTAL MODEL, FUNCTIONAL COMPONENTS, PROPERTY OF SELF-CONFIGURATION.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

P2P - Peer-to-Peer

SLO - service level objectives

DHT - Distributed Hash Table

Fractal ADL - Fractal architecture description language

VM - Virtual Machines

MAPE - Monitoring, Analyzing, Planning, Execution

LLC - limited Look-Ahead Control

CDN - Content Delivery Network

SNR - Set of Network References

YASS - Yet Another Storage Service

YACS - Yet Another Computing Service

ВСТУП

Актуальність дослідження. З появою кластеризованих і розподілених платформ як економічно ефективних середовищ паралельного програмування ряд критичних обчислень регулярно виконується саме через мережу. До них належать наукове моделювання, операції з аналізу даних і комерційні операції. Дані, що передаються через мережу в таких програмах, дуже сприйнятливі до перехоплення пакетів та інших атак, які порушують конфіденційність і цілісність обчислення.

У той час як проблема перехоплення пакетів може вирішуватися за допомогою безпечного тунелювання даних (шифрування всього зв'язку), накладні витрати на безпечне тунелювання чинить величезний тиск на і без того обмежену пропускну здатність зв'язку. Проблема захисту переданих даних набагато краще вирішується на прикладному рівні протоколів на відміну від загальних криптографічних протоколів.

В інших видах обчислень, суб'єкти, які беруть участь в паралельних програмах обчислень можуть не довіряти один одному. Наприклад, дві конкуруючі мережі можуть об'єднати зусилля у видобутку своїх даних про продажі. Проте, можливо, ніхто не захоче розкривати іншій стороні точні цифри продажів. Для більшості баз даних, операції приймають форму обчислення скалярного добутку з зовнішнім резидентним вектором бази даних. Ці вектори цінні для обох сторін, і витік інформації має певну ціну. Тим не менш, популярність інтелектуального аналізу даних вимагає компромісу між швидкістю та безпекою.

В інших типах середовищ можуть використовуватися два хости для виконання ненадійного протоколу обчислень, щоб захистити дані від можливості того, що один із субєктів може були скомпрометований. У таких випадках субєкти можуть погодитися брати участь у обчисленнях, якщо мають власні дані не скомпрометовані (розкриті) іншою стороною.

Крім недовіри до мережі та інших учасників, сторони підлягають захищеним та анонімним протоколам традиційних обмежень щодо паралельної обробки, а саме щодо прискорення паралелізму.

Маючи на увазі ці цілі, пропонується безпечний протокол для ключового комп'ютерного ядра. За своєю природою дот-добуток охоплює певну кількість інформації. Якщо дві сторони намагаються обчислити безпечний та анонімний скалярний добуток, одна зі сторін може виявити введення вектора іншої сторони, просто перевіривши його у відповідному місці. Завдяки цьому можна очікувати, що протокол програми вищого рівня заборонить велику кількість входжень на одному векторі. Тому слід сфокусуватися тут на захисті виділеного скалярного добутку за допомогою протоколу який має низькі обчислювальні (комунікаційні) витрати, хороші властивості безпеки та чудові числові показники стабільності.

Мета і задачі дослідження. **Метою магістерської роботи** є дослідження концепції та розкриття сутності моделей покращення керованості великих масштабованих розподілених обчислювальних систем.

Для досягнення поставленої мети необхідно розв'язати такі задачі:

- виконати аналіз методологій керування розподіленими обчислювальними системами ;
- структуризувати моделі самоналаштовуваного адаптивного керування системою;
- розробити структуру системи самоналаштовуваного адаптивного керування обчислювальною системою;
- виконати побудову моделі та шаблонів взаємодії розподіленого обчислювального середовища;
- виконати імплементацію концепції та моделей самокерованих додатків.

Об'єктом дослідження є моделі самоналаштовуваного адаптивного керування системою в контексті керованості великих розподілених систем.

Предметом дослідження є імплементація моделей покращення керованості великих масштабованих розподілених обчислювальних систем.

Методи дослідження базуються на використанні методів керування мережами, методів ранжування та впорядкування, моделей хмарних обчислень, методів та моделей побудови компонентних рішень та архітектур.

Наукова новизна одержаних результатів полягає у тому, що на основі ґрунтовного аналізу розподілених систем було побудовано моделі та шаблони взаємодії розподіленого обчислювального середовища в якому вирішено проблеми ефективності та надійності виявлення ресурсів, застосовано push-механізм, а не механізм витягування на основі децентралізованого підходу до управління.

Практичне значення одержаних результатів полягає в розробці структури керованості масштабованих систем яка дозволяє розробляти розподілені компонентні додатки із самоналаштовуваною системою керування, які не залежать від функціонального коду додатка, але можуть взаємодіяти з ним, коли це необхідно. Фреймворк забезпечує невеликий набір абстракцій, які сприяють надійному й ефективному управлінню додатками навіть у динамічних середовищах.

Апробація результатів дослідження. Матеріали дослідження було представлено у матеріалах І Всеукраїнської науково-практичної інтернет конференції “ІТ екосистема: цифровізація бізнес-процесів в умовах війни”, у тезах доповіді “Контроль ресурсів у високоефективних розподілених обчислювальних системах”.

Структура. Кількість розділів – 3. Загальний обсяг основної частини – 91 сторінки. Список використаних джерел містить – 52 позиції.

РОЗДІЛ 1. АНАЛІЗ МЕТОДОЛОГІЙ КЕРОВАНОСТІ ВЕЛИКИМИ РОЗПОДІЛЕНИМИ ОБЧИСЛЮВАЛЬНИМИ СИСТЕМАМИ

1.1 Опис концепції розподілених обчислювальних систем

Розподілені системи, такі як однорангові (P2P - Peer-to-Peer) [1], Grid [2] і хмарні [3] системи, забезпечують об’єднання та скоординоване використання розподілених ресурсів і послуг. Розподілені системи надають платформи для забезпечення великомасштабних розподілених програм, таких як обмін файлами P2P, багаторівневі програми Web 2.0 (наприклад, соціальні мережі та вікі) та наукові програми (наприклад, прогноз погоди та моделювання клімату). Зростаюча популярність і попит на великомасштабні розподілені програми прийшли зі збільшенням складності та накладних витрат на управління цими програмами, що створило проблему, яка перешкоджає подальшому розвитку [4]. Автономні обчислення [5] є привабливою парадигмою для вирішення проблеми зростаючої складності програмного забезпечення, роблячи програмні системи та програми самокерованими. Самоуправління, а саме самоконфігурація, самооптимізація, самовідновлення та захисту, можна досягти за допомогою автономних менеджерів [6]. Автономний менеджер постійно контролює програмне забезпечення та середовище його виконання та діє для досягнення цілей управління, таких як відновлення після збою або оптимізація продуктивності. Управління програмами в динамічних середовищах з динамічними ресурсами та/або навантаженням (як-от спільнота Grids, однорангові системи та хмари) є особливо складним через великий масштаб, складність, високий відтік ресурсів (наприклад, у системах P2P) і відсутність чіткої відповідальності керівництва.

Більшість розподілених систем і додатків побудовані з компонентів з використанням моделей розподілених компонентів, таких як Fractal

Component Model [7] і Kompics Component Framework [8], тому ми вважаємо, що самокерування має бути включено на рівні компонентів, щоб підтримувати моделі розподілених компонентів для розробки великомасштабних динамічних розподілених систем і додатків.

Ці розподілені програми повинні керувати собою, маючи певні властивості самоналаштування (тобто самовідновлення, самозахист, самооптимізація), щоб виживати у високодинамічному розподіленому середовищі та забезпечувати необхідну функціональність у прийнятний рівень продуктивності. Властивості Self можна надати за допомогою циклів керування зворотним зв’язком, відомих як цикли MAPE-K (монітор, аналіз, план, виконання – знання), які походять із сфери автономних обчислень . Першим кроком на шляху до самоуправління у великомасштабних розподілених системах є надання розподілених служб зондування та керування, які самі по собі є самокерованими. Іншим важливим кроком є забезпечення надійної абстракції керування, яку можна використовувати для побудови циклів MAPE-K. Ці послуги та абстракції повинні забезпечувати надійні гарантії якості обслуговування в умовах відтоку та еволюції системи.
Суть нашого підходу до самоуправління для розподілених систем базується на використанні властивостей самоорганізації структурованих накладених мереж для надання базових послуг і підтримки виконання разом із моделями компонентів для реконфігурації та самоаналізу. Кінцевим результатом є автономна обчислювальна платформа, придатна для великомасштабних динамічних розподілених середовищ. Структуровані накладені мережі розроблені для роботи у високодинамічному розподіленому середовищі, на яке ми орієнтуємося. Вони мають певні властивості самозахисту і можуть терпіти відтік. Таким чином, структуровані накладні мережі можна використовувати як основу для підтримки самоуправління в розподіленій системі, наприклад, як середовище зв’язку (для передачі повідомлень, трансляції та маршрутизації), пошуку (розподілені хеш-таблиці та зв’язок на основі імен) і послуга публікації/підписки.

Щоб краще справлятися з динамічним середовищем; покращити масштабованість, надійність і продуктивність, уникаючи вузьких місць управління та єдиної точки відмови; ми виступаємо за розподіл функцій управління між декількома менеджерами кооперативу, які координують свою діяльність для досягнення цілей управління.

Деякі проблеми виникають під час спроби ввімкнути самокерування для великих складних розподілених систем, які не з'являються в централізованих і кластерних системах. Ці проблеми включають тривалі затримки мережі та труднощі підтримки глобальних знань про систему. Ці проблеми впливають на спостережливість і керованість системи управління і можуть перешкодити нам безпосередньо застосовувати класичні теорії керування для побудови контурів керування. Іншим важливим питанням є координація між декількома автономними менеджерами, щоб уникнути конфліктів і коливань. Автономні менеджери також повинні бути відтворені в динамічних середовищах, щоб терпіти збої.

Зростаюча популярність додатків Web 3.0, таких як вікі, соціальні мережі та блоги, поставила нові виклики для основної інфраструктури забезпечення. Багато великомасштабних програм Web 2.0 використовують еластичні служби, такі як еластичні сховища ключів і значень, які можна масштабувати горизонтально шляхом додавання/видалення серверів. Voldemort [9], Cassandra [10] і Dynamo [11] є кількома прикладами послуг еластичного зберігання. Хмарні обчислення [3] з моделлю оплати за використання є привабливим середовищем для надання еластичних послуг, оскільки поточні витрати на такі послуги стають пропорційними кількості ресурсів, необхідних для обробки поточного навантаження.

Управління ресурсами для програм Web 3.0, щоб гарантувати прийнятну продуктивність, є складним завданням, оскільки важко передбачити навантаження, особливо для нових програм, які можуть стати популярними протягом кількох днів [12,13]. Крім того, вимоги до

продуктивності зазвичай виражаються у вигляді верхніх процентилів, які важче підтримувати цю середню продуктивність [11,14].

Розрахункова модель ціноутворення, еластичність і динамічне робоче навантаження разом викликають потребу в контролері еластичності, який автоматизує надання хмарних ресурсів. Контролер еластичності додає більше ресурсів під високим навантаженням, щоб досягти необхідних цілей рівня обслуговування (SLO - service level objectives), і звільняє деякі ресурси під низьким навантаженням, щоб зменшити витрати.

1.2 Концепція та сутність автономних обчислень

У 2001 році Пол Хорн з IBM ввів термін автономні обчислення, щоб відзначити початок нової парадигми обчислень [5]. Автономні обчислення зосереджені на вирішенні проблеми зростаючої складності програмного забезпечення. Ця проблема становить серйозний виклик як для науки, так і для промисловості, оскільки зростаюча складність обчислювальних систем ускладнює для IT-персоналу розгортання, керування та підтримку таких систем. Це різко збільшує витрати на управління. Крім того, за відсутності належного та своєчасного керування продуктивність системи може впасти або система навіть може вийти з ладу. Іншим недоліком збільшення складності є те, що воно змушує нас більше зосереджуватися на вирішенні питань управління замість того, щоб вдосконалювати саму систему та рухатися вперед до нових інноваційних програм.

Вегетативні обчислення створені завдяки вегетативній системі, яка постійно підсвідомо регулює та захищає наше тіло [17], залишаючи нам можливість зосередитися на іншій роботі. Подібним чином автономна система повинна знати про своє оточення, постійно контролювати себе та адаптуватися відповідно з мінімальною участю людини. Менеджери-люди повинні вказувати лише політики вищого рівня, які визначають загальну поведінку системи. Це зменшить витрати на управління, покращить

продуктивність і дозволить розробляти нові інноваційні програми. Мета автономних обчислень полягає не в тому, щоб повністю замінити адміністраторів-людей, а радше в тому, щоб дозволити системам автоматично налаштовуватися та адаптуватися відповідно до політики, що розвивається, визначеної людьми.

1.2.1 Властивості самокерованих систем

IBM запропонувала основні властивості, які повинна мати будь-яка самокерована система [4], щоб бути автономною системою. Ці властивості зазвичай називають властивостями self. Чотири основні властивості:

- Самоналаштування: автономна система повинна мати можливість самоналаштуватися на основі поточного середовища та доступних ресурсів. Система також повинна мати можливість постійно змінювати конфігурацію та адаптуватися до змін.
- Самооптимізація: система повинна постійно контролювати себе та намагатися налаштувати себе та підтримувати продуктивність (та/або інші робочі показники, такі як споживання енергії та вартість) на оптимальному рівні.
- Самовідновлення: збої повинні виявлятися системою. Після виявлення система повинна мати можливість відновлюватися після збою та виправляти себе.
- Самозахист: система повинна мати можливість захистити себе від зловмисного використання. Це включає, наприклад, захист від вірусів і спроб вторгнення.

1.2.2 Архітектура автономного обчислення

Еталонна архітектура автономних обчислень, запропонована IBM [6], складається з наступних п’яти складових (рис . 1.1).

Touchpoint складається з набору датчиків і ефекторів (приводів), які використовуються автономними менеджерами для взаємодії з керованими ресурсами (отримання статусу та виконання операцій). Точки дотику – це компоненти системи, які реалізують уніфікований інтерфейс керування, який приховує неоднорідність керованих ресурсів. Керований ресурс має бути представлений через точки дотику щоб він був керованим. Датчики дають інформацію про стан ресурсу. Ефектори забезпечують набір операцій, які можна використовувати для зміни стану ресурсів.

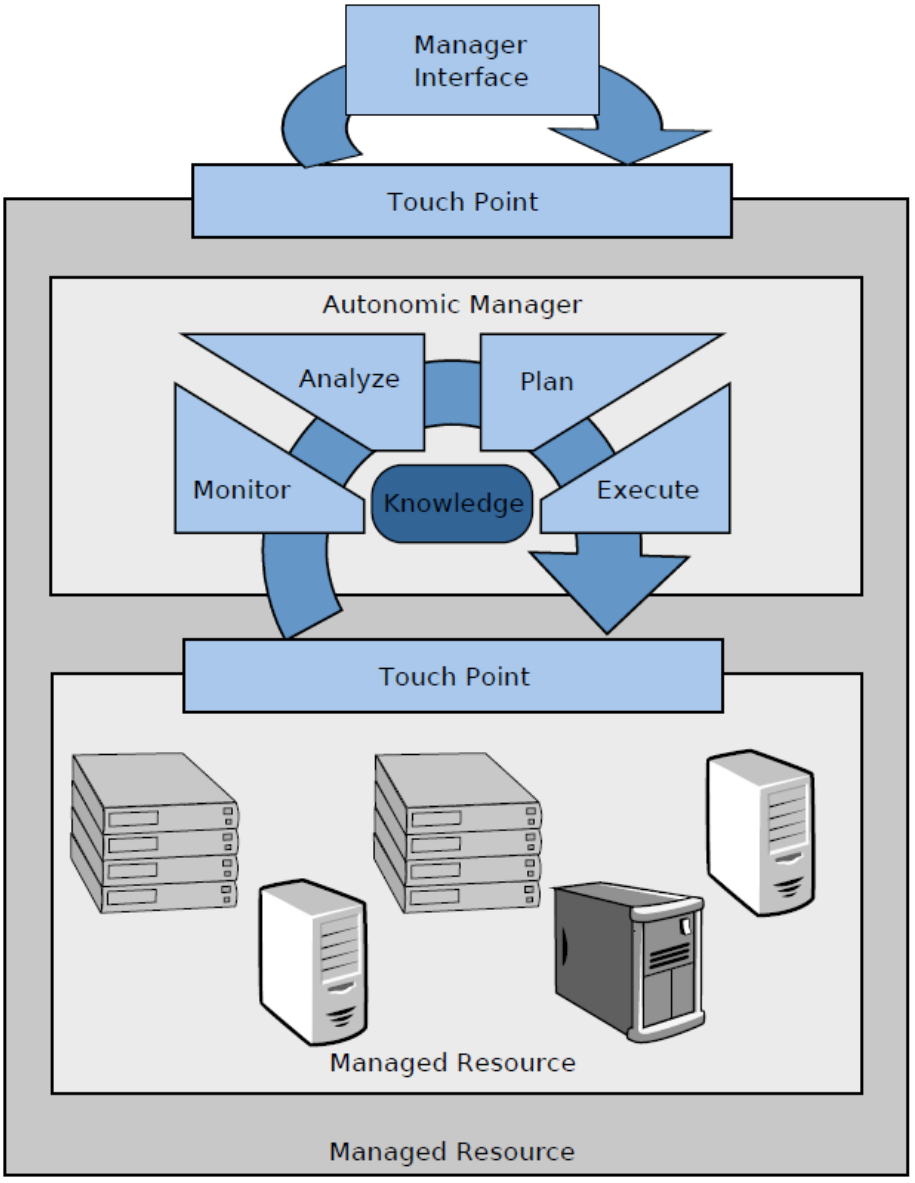


Рисунок 1.1 – Проста автономна обчислювальна архітектура

Autonomic Manager є ключовим елементом архітектури. Автономні менеджери використовуються для реалізації поведінки самоуправління системи. Це досягається за допомогою циклу керування, який складається з чотирьох основних етапів: моніторинг, аналіз, планування та виконання. Контур керування взаємодіє з керованим ресурсом через відкриті точки дотику.

Джерело знань використовується для обміну знаннями (наприклад, інформацією про архітектуру, історією моніторингу, політиками та даними керування, такими як плани змін) між автономними менеджерами.

Enterprise Service Bus забезпечує підключення компонентів системи. Manager Interface надає адміністраторам інтерфейс для взаємодії з системою. Це включає в себе можливість відстежувати/змінювати статус системи та контролювати автономні менеджери за допомогою політик.

У нашій роботі ми пропонуємо дизайн і реалізацію еталонної архітектури автономних обчислень для великомасштабних розподілених систем.

1.2.3 Підходи до автономних обчислень

Останні дослідження прийняли різні підходи до досягнення автономної поведінки в обчислювальних системах. Найпопулярніші підходи описані нижче.

Архітектурний підхід: цей підхід підтримує створення автономних систем із компонентів. Це тісно пов'язане з сервіс-орієнтованими -архітектурами. Властивості компонентів, включаючи необхідні інтерфейси, очікувану поведінку, встановлення взаємодії та шаблони проектування [18]. Автономна поведінка обчислювальних систем досягається за допомогою динамічної модифікації структури (композиційної адаптації) і, таким чином, поведінки системи [19, 20] у відповідь на зміни в середовищі або поведінці

користувача. Управління в цьому підході здійснюється на рівні компонентів і взаємодії між ними.

Теоретичний підхід до керування: класична теорія управління успішно застосовувалася для вирішення проблем керування в обчислювальних системах [21], таких як балансування навантаження, регулювання пропускної здатності та керування живленням. Концепції та методи теорії управління використовуються для керівництва розробкою автономних менеджерів для сучасних самокерованих систем [22]. Виклики, що виходять за межі класичної теорії управління, також були розглянуті [23], такі як використання проактивного керування (модельного прогнозного керування) для подолання мережевих затримок і невизначеності операційних середовищ, а також багатопараметрична оптимізація в дискретній області.

Підхід, заснований на виникненні: Цей підхід де складні структури або поведінка виникають у результаті відносно простих взаємодій. В обчислювальних системах загальна автономна поведінка системи на макрорівні безпосередньо не програмується, а виникає з відносно простої поведінки різних підсистем на мікрорівні [24-26]. Цей підхід є дуже децентралізованим. Підсистеми приймають рішення автономно на основі своїх локальних знань і погляду на систему. Зв'язок зазвичай простий, асинхронний і використовується для обміну даними між підсистемами.

Підхід на основі агентів: на відміну від традиційних підходів до управління, які зазвичай є централізованими або ієрархічними, підхід до управління на основі агентів є децентралізованим. Це підходить для великомасштабних обчислювальних систем, які розподілені з багатьма складними взаємодіями. Агенти в багатоагентній системі співпрацюють, координують і домовляються один з одним, утворюючи суспільство або організацію для вирішення проблеми розподіленого характеру [27,28].

Застарілі системи: Дослідження в цій галузі намагаються додати -способи самокерування до вже існуючих (застарілих) систем. Дослідження

включають методи моніторингу та активації застарілих систем, а також визначення вимог до керованості систем [29-32].

У нашій роботі над розподіленою системою керування компонентами ми дотримувалися в основному архітектурного підходу до автономних обчислень. Ми використовуємо та розширюємо фрактальну компонентну модель, щоб динамічно змінювати структуру і, таким чином, поведінку системи. Однак чіткої межі між цими різними підходами немає, і їх можна об’єднати в одну систему. Пізніше, у нашому дослідженні автоматизації еластичності для хмарних сервісів, ми використали підхід теорії контролю до самоуправління.

1.3 Дослідження фрактальної компонентної моделі

Фрактальна компонентна модель [7, 33] є модульною та розширюваною компонентною моделлю, яка використовується для проектування, реалізації, розгортання та реконфігурації різних систем і додатків. Фрак тал не залежить від мови програмування та моделі виконання. Основна мета фрактальної компонентної моделі полягає в тому, щоб зменшити витрати на розробку, розгортання та обслуговування складних програмних систем. Це досягається в основному за рахунок поділу проблем, які виникають на різних рівнях, а саме: поділ інтерфейсу та реалізації, компонентно-орієнтоване програмування та інверсія керування. Розділення інтерфейсу та реалізації відокремлює дизайн від реалізації. Компонентно-орієнтоване програмування розділяє реалізацію на більш дрібні відокремлені завдання, призначені компонентам. Інверсія контролю розділяє функціональні та управлінські проблеми.

Компонент у фракталі складається з двох частин: мембрани та вмісту. Мембрана відповідає за нефункціональні властивості компонента, тоді як вміст відповідає за функціональні властивості. Доступ до компонента можна отримати через інтерфейси. Існує три типи інтерфейсів: клієнт, сервер і

інтерфейс керування. Клієнтський і серверний інтерфейси можуть бути пов’язані між собою за допомогою прив’язок, тоді як інтерфейси керування використовуються для контролю та інтроспективи компонента. Фрактальний компонент може бути основою композитного компонента. У випадку базового компонента зміст є безпосередньою реалізацією його функціональних властивостей. Вміст складеного компонента складається з кінцевого набору інших компонентів. Таким чином, програма Fractal складається з набору компонентів, які взаємодіють через композицію та прив’язки.

Фрактал дозволяє керувати складними програмами, роблячи архітектуру програмного забезпечення явною. Головним чином це пов’язано з рефлексивністю компонентної моделі, що означає, що компоненти мають повні можливості самоаналізу (через інтерфейси керування). Основними контролерами, визначеними в фракталі, є контроль атрибутів, контроль прив’язки, контроль вмісту та контроль життєвого циклу.

Рисунок 1.2 ілюструє різні сутності типової архітектури фрактального компонента. Товсті чорні поля позначають контролерну частину компонента, тоді як внутрішня частина коробок відповідає вмістовій частині компонента.

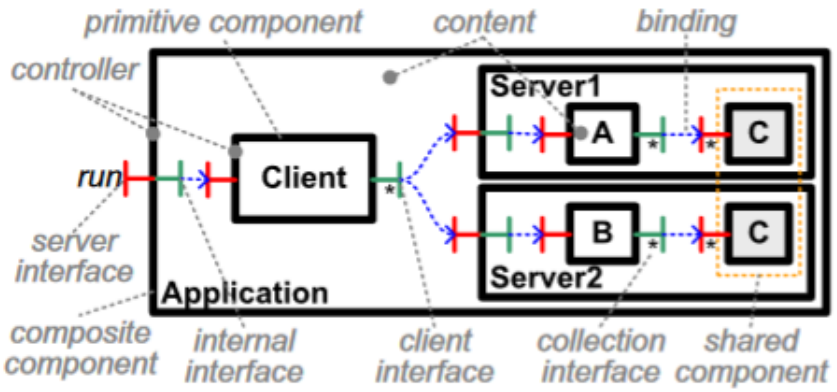


Рисунок 1.2 – Фрактальна компонентна модель

Стрілки відповідають прив’язкам, а тауподібні структури, що виступають із чорних ящиків, є внутрішніми або зовнішніми інтерфейсами. Внутрішні інтерфейси доступні лише з вмісту компонента. Зірчастий

інтерфейс представляє набір інтерфейсів одного типу. Дві заштриховані рамки C представляють спільний компонент.

Модель також включає мову опису фрактальної архітектури (Fractal ADL - Fractal architecture description language), яка є XML-документом, який використовується для опису фрактальної архітектури додатків, включаючи опис компонентів (інтерфейси, реалізація, мембрана тощо) і зв'язок між компонентами (композиція та прив'язки). Fractal ADL також можна використовувати для розгортання програми Fractal, де аналізатор ADL аналізує файл ADL програми та створює екземпляри відповідних компонентів і прив'язок.

У нашій роботі ми використовуємо фрактальну компонентну модель для самоаналізу та реконфігурації компонентів розподіленої програми. Ми розширюємо компонентну модель різними способами, такими як мережеві прозорі зв'язки, які забезпечують мобільність компонентів, а також за допомогою груп компонентів і зв'язків «один до будь-кого» та «один до всіх».

1.4 Структуровані однорангові накладені мережі

Однорангова мережа (P2P) відноситься до класу розподілених мережевих архітектур, які утворюються з учасників (зазвичай званих одноранговими вузлами), які знаходяться на межі Інтернету. P2P стає все більш популярним, оскільки периферійні пристрої стають все більш потужними з точки зору підключення до мережі, зберігання та процесорної потужності. Загальною особливістю всіх P2P-мереж є те, що учасники утворюють співтовариство однорангових партнерів, де одноранговий партнер спільноти ділиться деякими ресурсами (наприклад, сховищем, пропускною здатністю або обчислювальною потужністю) з іншими, а натомість він може використовувати ресурси, спільні для інших. [1]. Іншими словами, кожен вузол виконує роль і клієнта, і сервера. Таким чином, мережа P2P зазвичай не потребує центрального сервера і працює в децентралізований спосіб. Іншою

важливою особливістю є те, що однорангові вузли також відіграють роль маршрутизаторів і беруть участь у маршрутизації повідомлень між одноранговими вузлами в накладенні.

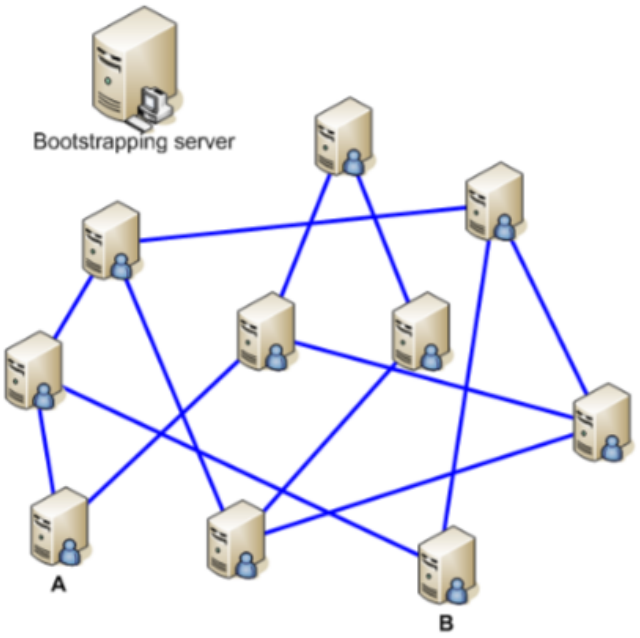


Рисунок 1.3 – Децентралізовані неструктуровані однорангові мережі

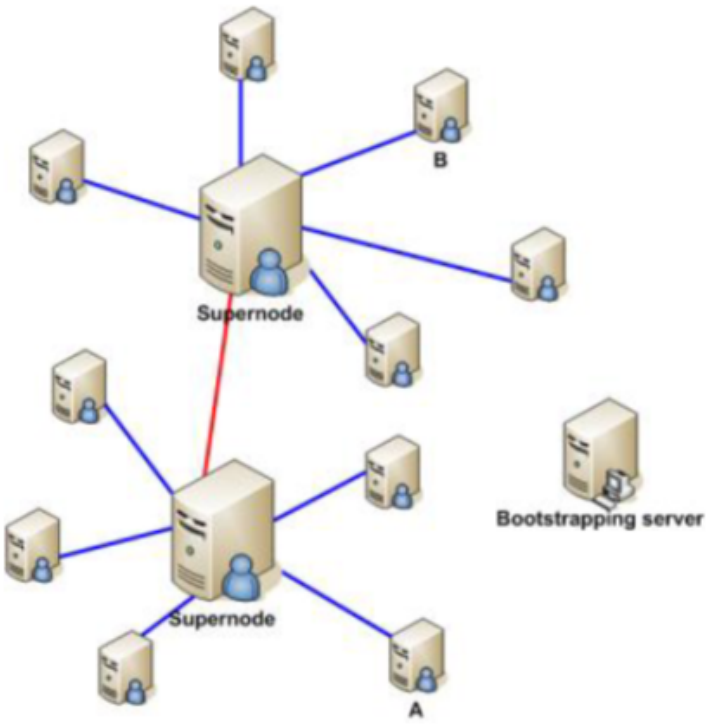


Рисунок 1.4 – Гібридні неструктуровані однорангові мережі

Мережі P2P є масштабованими та надійними. Той факт, що кожен вузол відіграє роль як клієнта, так і сервера, має значний вплив на можливість масштабування мереж P2P до великої кількості вузлів. Це пояснюється тим, що, на відміну від традиційної клієнт-серверної моделі, додавання більшої кількості однорангових вузлів збільшує ємність системи (наприклад, додає більше пам'яті та пропускної здатності). Ще один важливий фактор, який сприяє масштабуванню P2P, полягає в тому, що вузли діють як маршрутизатор. Таким чином, кожному однолітку потрібно знати лише про підмножину інших однорангових пристроїв. Децентралізований характер мереж P2P покращує їх надійність. Немає єдиної точки відмови, і P2P-мережі розроблені таким чином, щоб однорангові мережі могли приєднуватися, залишати та виходити з ладу в будь-який момент.

Однорангові вузли в мережі P2P зазвичай утворюють накладену мережу поверх фізичної топології мережі. Накладення складається з віртуальних зв'язків, які встановлюються між одноранговими вузлами певним чином відповідно до типу (топології) мережі P2P. Віртуальне з'єднання між будь-якими двома одноранговими вузлами в накладенні може бути реалізоване декількома з'єднаннями у фізичній мережі. Накладення зазвичай використовується для зв'язку, індексування та виявлення однорангових користувачів. Спосіб формування посилань у накладенні поділяє мережі P2P на два основні класи: неструктуровані та структуровані мережі. Накладені зв'язки між одноранговими вузлами в неструктурованій мережі P2P формуються випадковим чином без будь-якого алгоритму організації структури. З іншого боку, накладені зв'язки між одноранговими вузлами в структурованій мережі P2P дотримуються фіксованої структури, яка постійно підтримується алгоритмом. Решта цього розділу буде зосереджена на структурованих мережах P2P.

Структурована мережа P2P, така як Chord [34], CAN [35] і Pastry [36] , підтримує структуру накладених посилань. Використання цієї структури дозволяє реалізувати розподілену хеш-таблицю (DHT). DHT надає службу

пошуку, подібну до хеш-таблиць, яка зберігає пари ключ-значення. Отримавши ключ, будь-який вузол може ефективно отримати пов’язане значення шляхом маршрутизації запиту відповідальному вузлу. Відповідальність за підтримку відповідності між парами ключ-значення та інформацією про маршрутизацію розподіляється між одноранговими вузлами таким чином, що приєднання/вихід/збій однорангового вузла спричиняє мінімальні збої в службі пошуку. Це технічне обслуговування відбувається автоматично і не вимагає участі людини. Ця функція відома як самоорганізація.

Більш складну послугу можна створити на основі DHT. Такі послуги включають зв’язок на основі імен, ефективну групову/широкомовну передачу, послуги публікації/підписки та розподілені файлові системи.

У нашій роботі ми використовували структуровані накладні мережі та сервіси, побудовані на них, як середовище зв’язку між різними компонентами системи (функціональними компонентами та елементами керування). Ми використовуємо масштабованість і властивості самоорганізації (наприклад, автоматичне виправлення таблиць маршрутизації для того, щоб терпіти приєднання, вихід і відмови однорангових пристроїв, автоматичне збереження відповідальності за сегменти DHT) структурованої мережі P2P для надання основних послуг і підтримки виконання. Ми використовували службу індексування, щоб реалізувати прозорий мережевий зв’язок на основі імен і груп компонентів. Ми використовували ефективну багатоадресну трансляцію для спілкування та виявлення. Ми використовували службу публікації/підписки, щоб реалізувати зв’язок між елементами керування на основі подій.

1.5. Огляд сучасних мережевих сервісів

Зростаюча популярність мережевих додатків, таких як вікі, соціальні мережі та блоги, поставила нові виклики для основної інфраструктури

забезпечення. Дані програми орієнтовані на дані з частим доступом до даних [37]. Це створює нові проблеми для рівня даних n-рівневих серверів додатків, оскільки продуктивність рівня даних зазвичай регулюється суворими цілями рівня обслуговування (SLO - Service Level Objectives) [14], щоб задовольнити очікування клієнтів.

Із швидким зростанням кількості користувачів користувачів Web погана масштабованість типового рівня даних із властивостями ACID [38] обмежувала масштабованість програм Web 2.0. Це призвело до розробки нових сховищ даних із послабленими гарантіями узгодженості та простіших операцій, таких як Voldemort [9], Cassandra [10] і Dynamo [11]. Ці системи зберігання зазвичай забезпечують просте сховище ключ-значення з кінцевими гарантіями узгодженості. Спрощені моделі даних і узгодженості сховищ ключів і значень дозволяють ефективно масштабувати їх горизонтально, додаючи більше серверів і таким чином обслуговуючи більше клієнтів.

Ще одна проблема, з якою стикаються додатки Web 2.0, полягає в тому, що певна служба, функція або тема можуть раптово стати популярними, що призведе до різкого збільшення робочого навантаження [12,13]. Той факт, що сховище є службою зі збереженням стану, ускладнює проблему, оскільки лише певна підмножина серверів розміщує дані, пов’язані з популярним елементом. Підмножина стає перевантаженою, тоді як інші сервери можуть бути недовантаженими.

Ці виклики призвели до необхідності автоматизованого підходу до керування рівнем даних, здатного швидко й ефективно реагувати на зміни робочого навантаження, щоб відповідати необхідним SLO служби зберігання.

1.5.1 Хмарні обчислення та еластичні послуги

Хмарні обчислення [3] з моделлю оплати за використання є привабливим рішенням для розміщення постійно зростаючої кількості додатків Web 2.0, оскільки експлуатаційна вартість таких послуг стає

пропорційною кількості ресурсів, необхідних для обробки поточне навантаження. Ця модель є привабливою, особливо для стартапів, тому що важко передбачити майбутнє навантаження, яке буде накладено на додаток, і, отже, кількість ресурсів (наприклад, серверів), необхідних для обслуговування цього навантаження. Іншою причиною є початкові інвестиції у формі купівлі серверів, яких уникають завдяки моделі ціноутворення Cloud Pay-A-You-Go. Незалежність пікових навантажень для різних програм дозволяє хмарним провайдерам ефективно розподіляти ресурси між програмами. Однак спільне використання фізичних ресурсів між віртуальними машинами (VM), які запускають різні програми, ускладнює моделювання та прогнозування продуктивності ВМ [39,40].

Щоб використовувати хмарну модель ціноутворення та ефективно справлятися з динамічним робочим навантаженням Web 2.0, хмарні служби (такі як сховище ключ-значення на рівні даних хмарної багаторівневої програми) розроблені, щоб бути еластичними. Еластична служба розроблена таким чином, щоб мати можливість масштабуватись горизонтально під час виконання, не перериваючи запущену службу. Еластичний сервіс може бути збільшений (наприклад, системним адміністратором) у разі збільшення робочого навантаження, додавши більше ресурсів, щоб відповідати SLO. У разі зменшення навантаження еластичну послугу можна зменшити шляхом видалення додаткових ресурсів і, таким чином, зниження вартості без порушення SLO. Для служб із збереженням стану масштабування зазвичай поєднується з кроком відновлення балансу, необхідним для перерозподілу даних між новим набором серверів.

Управління ресурсами для додатків Web 2.0, щоб гарантувати прийнятну продуктивність, є складним завданням через поступові (щоденні) і раптові зміни робочого навантаження [41]. Важко передбачити робоче навантаження, особливо для нових програм, які можуть стати популярними протягом кількох днів [12,13]. Крім того, вимоги до продуктивності зазвичай виражаються у вигляді верхніх процентилів (наприклад, «99% зчитувань

виконуються менш ніж за 10 мс протягом однієї хвилини»), які важче підтримувати, ніж середню продуктивність [11,14].

Розрахункова модель ціноутворення, еластичність і динамічне робоче навантаження програм Web 2.0 разом викликають потребу в контролері еластичності, який автоматизує надання хмарних ресурсів. Контролер еластичності використовує горизонтальну масштабованість еластичних служб, надаючи більше ресурсів за високих навантажень, щоб досягти необхідних цілей рівня обслуговування (SLO). Розрахункова модель ціноутворення забезпечує стимул для контролера еластичності вивільняти додаткові ресурси, коли вони не потрібні, коли робоче навантаження зменшується.

В обчислювальних системах контролер [21] або автономний менеджер [5] – це компонент програмного забезпечення, який регулює нефункціональні властивості (показники продуктивності) цільової системи. Нефункціональні властивості – це такі властивості системи, як час відгуку або використання ЦП. З точки зору контролера, ці показники продуктивності є результатом системи. Регулювання досягається шляхом моніторингу цільової системи через інтерфейс моніторингу та адаптації конфігурацій системи, таких як кількість серверів, відповідно через інтерфейс керування (контрольний вхід). Контролери можна класифікувати на контролери зі зворотним зв’язком або з прямим зв’язком залежно від того, чи використовує контролер зворотний зв’язок для керування системою. Контроль із зворотним зв’язком вимагає моніторингу виходу системи, тоді як керування з випереджальним зв’язком не відстежує вихід системи, оскільки він не використовує вихід для керування.

При управлінні зворотним зв'язком вихідні дані системи (наприклад, час відгуку) контролюються. Контролер обчислює похибку керування, порівнюючи поточний вихід системи з бажаним значенням, встановленим системними адміністраторами. Залежно від величини та знаку помилки керування, контролер змінює вхідний сигнал керування (наприклад, кількість

серверів, які потрібно додати або видалити), щоб зменшити помилку керування. Основна перевага управління зі зворотним зв'язком полягає в тому, що контролер може адаптуватися до збурень, таких як зміни в поведінці системи або її робочому середовищі.

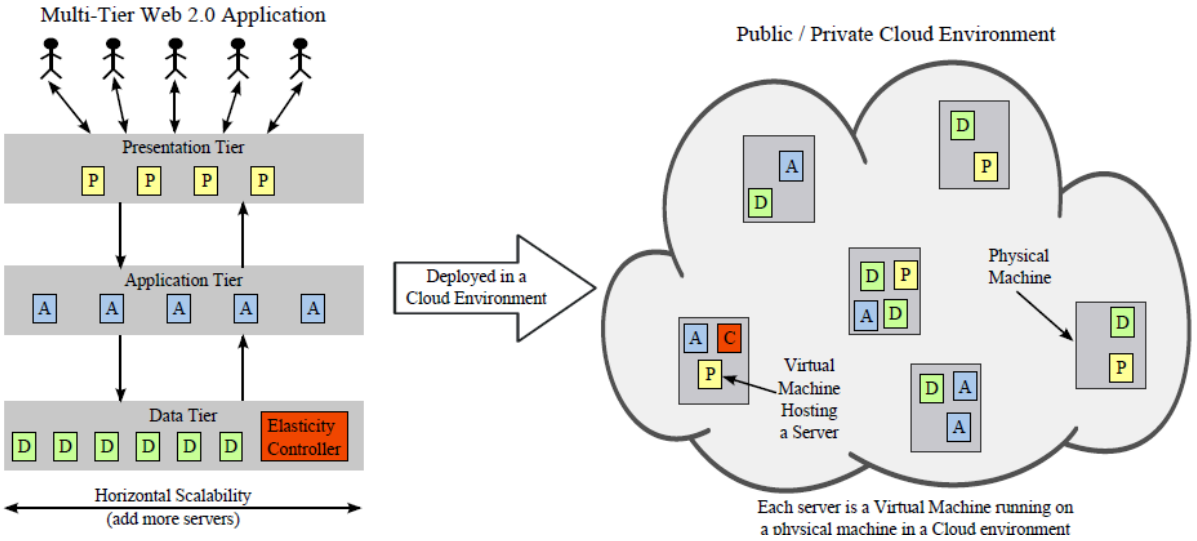


Рисунок 1.5 – Багаторівнева програма Web 2.0 з контролером еластичності, розгорнута в хмарному середовищі

Недоліки включають коливання, перерегулювання та можливу нестабільність, якщо контролер не сконструйований належним чином. Через нелінійність більшості систем регулятори зі зворотним зв'язком апроксимуються навколо лінійних областей, які називаються робочою областю. Контролери зворотного зв’язку працюють належним чином лише в робочому регіоні, для якого вони розроблені.

У прямому управлінні вихідні дані системи не контролюються. Натомість контролер прямого зв’язку покладається на модель системи, яка використовується для обчислення вихідних даних системи на основі поточного стану системи. Наприклад, враховуючи поточну швидкість запитів і кількість серверів, модель системи використовується для розрахунку відповідного часу відповіді та дій відповідно до бажаного часу відповіді. Головним недоліком прямого керування є те, що воно дуже чутливе до

неочікуваних збурень, які не враховані в системній моделі. Зазвичай це призводить до відносно складної моделі системи порівняно з керуванням зі зворотним зв’язком. Основні переваги прямого керування включають швидшість, ніж керування зі зворотним зв’язком, а також уникнення коливань і перерегулювання.

Висновки до розділу 1

В даному розділі проведено аналіз методологій керування розподіленими обчислювальними системами, описано концепції побудови розподілених систем та автономних обчислень. Також приведена фрактальна компонентна модель, описані структуровані однорангові накладені мережі та приведені основні переваги мережевих та хмарних сервісів.

РОЗДІЛ 2. СТРУКТУРА МОДЕЛІ САМОНАЛАШТОВУВАНОГО АДАПТИВНОГО КЕРУВАННЯ РОЗПОДІЛЕНОЮ ОБЧИСЛЮВАЛЬНОЮ СИСТЕМОЮ

2.1 Сучасний стан справ в області самоуправління великих розподілених систем

Необхідно зменшити вартість володіння програмним забезпеченням, тобто вартість адміністрування, управління, обслуговування та оптимізації програмних систем, а також мережевих середовищ, таких як Grids, Clouds і P2P-системи. Ця потреба викликана неминучим зростанням складності та масштабу програмних систем і мережевих середовищ, які стають надто складними, щоб ними безпосередньо керувати люди. Для багатьох таких систем ручне керування є складним, дорогим, неефективним і схильним до помилок.

Великомасштабна система може складатися з тисяч елементів, які необхідно відстежувати та контролювати, і мати велику кількість параметрів, які потрібно налаштувати, щоб оптимізувати продуктивність системи та потужність, покращити використання ресурсів та усунути несправності під час надання послуг відповідно до SLA. Найкращий спосіб обробки складності системи, адміністрування та експлуатаційних витрат полягає в розробці вегетативних систем, здатних керувати собою, як вегетативна система регулює та захищає організм людини [4,17]. Системне самокерування дозволяє зменшити витрати на управління та підвищити ефективність управління шляхом усунення адміністраторів-людей від більшості (низького рівня) механізмів управління системою, таким чином, основним обов’язком людей є визначення політики для автономного управління, а не керування механізмами, які реалізувати політику.

Зростаюча складність програмних систем і мережевих середовищ спонукає до дослідження автономних [4,5,17,45]. Великі постачальники комп'ютерів і програмного забезпечення започаткували науково-дослідні ініціативи в галузі автономних обчислень.

Основною метою дослідження автономної системи є автоматизація більшості функцій керування системою, включаючи керування конфігураціями, управління несправностями, керування продуктивністю, керування живленням, управління безпекою, управління витратами та керування SLA. Цілі самоуправління зазвичай класифікуються за чотирма категоріями: самоконфігурація, самовідновлення, самооптимізація та самозахист [4]. Основні цілі самоуправління у великомасштабних системах, таких як Clouds, включають усунення збоїв, покращення використання ресурсів, оптимізацію продуктивності, оптимізацію потужності, управління змінами (оновленнями). Автономне управління SLA також входить до списку завдань самоуправління. В даний час дуже важливо зробити самоуправління потужним, тобто мінімізувати споживання енергії при досягненні цілей рівня обслуговування [46].

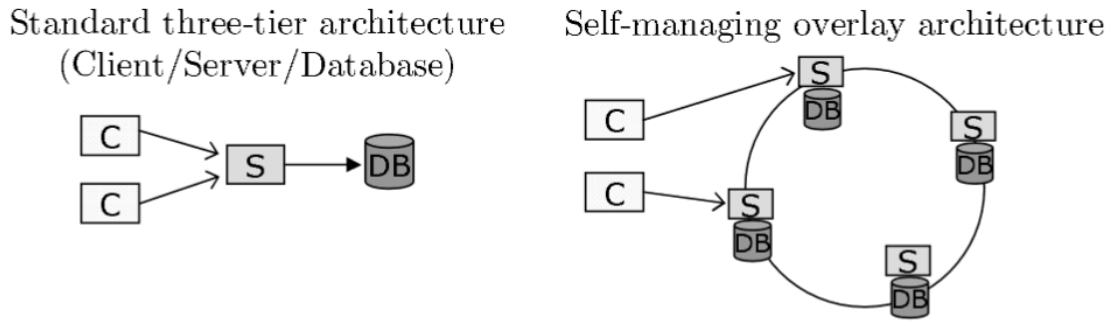


Рисунок 2.1 – Традиційна та самокерована трирівнева архітектура

Основний підхід до самоуправління полягає у використанні одного або кількох контурів керування зі зворотним зв'язком [17, 21], так званих автономних менеджерів [4], для керування різними властивостями системи на основі функціональної декомпозиції завдань управління та призначення

завдань декільком керівникам [47-49]. Кожен менеджер має конкретну мету управління (наприклад, оптимізація потужності або оптимізація продуктивності), яка може бути одного або комбінації трьох видів: регуляторний контроль (наприклад, підтримка використання сервера на певному рівні), оптимізація (наприклад, енергоспоживання). та оптимізації продуктивності), усунення завад (наприклад, забезпечення роботи під час модернізації системи) [21]. Керівний цикл керування складається з чотирьох етапів, відомих як MAPE (Monitoring, Analyzing, Planning, Execution): моніторинг, аналіз, планування та виконання [4].

Автори [21] застосовують теоретичний підхід до проектування обчислювальних систем зі зворотним зв'язком. Архітектурний підхід до автономних обчислень [18] пропонує визначити інтерфейси, вимоги до поведінки та шаблони взаємодії для архітектурних елементів, наприклад, компонентів. Етапи аналізу та планування циклу керування можуть бути реалізовані за допомогою функцій корисності для прийняття управлінських рішень, наприклад, для досягнення ефективного розподілу ресурсів [51]. Автори [49] і [48] використовують багатокритеріальні функції корисності для управління продуктивністю з урахуванням потужності. Автори [52, 53] використовують техніку модельно-прогнозного керування, а саме обмежене прогнозне керування (LLC - limited look-ahead control) у поєднанні з менеджерами на основі правил, щоб оптимізувати продуктивність системи на основі її прогнозованої поведінки в перспективі.

Автори [38] пропонують загальний протокол для динамічного розподілу ресурсів у великомасштабному хмарному середовищі, екземпляри якого можна створювати для конкретних цілей, за умов обмежень процесора та пам'яті. Автори ілюструють екземпляр загального протоколу, спрямованого на мінімізацію енергоспоживання за допомогою консолідації серверів, одночасно задовольняючи мінливий шаблон навантаження. Протокол мінімізує енергоспоживання завдяки консолідації серверів, коли система перебуває в недостатньому навантаженні, і використовує

справедливий розподіл ресурсів у разі перевантаження. Автори виступають за використання протоколу пліток для ефективного обчислення матриці конфігурації, яка визначає, як розподіляються ресурси хмари, для великомасштабних хмар.

Автори [39] розглядають проблему автоматизації горизонтальної еластичності хмарного сервісу, щоб задовольнити різноманітні вимоги до сервісу при дотриманні SLA. Автори використовують теорію масового обслуговування для моделювання хмарного сервісу. Модель використовується для створення двох адаптивних проактивних контролерів, які оцінюють майбутнє навантаження на сервіс. Автори пропонують використовувати гібридний контролер, що складається з проактивного контролера для масштабування в поєднанні з реактивним контролером для масштабування.

Самоуправління на основі політики [43,44] дозволяє високорівневу специфікацію цілей управління у формі політик , які керують автономним управлінням і можуть бути змінені під час виконання. Управління на основі політики можна поєднати з «жорстко закодованим» управлінням.

Існує кілька промислових рішень (інструментів, методів і пакетів програмного забезпечення) для забезпечення та досягнення самостійного керування корпоративними IT-системами, наприклад Tivoli від IBM та OpenView від HP, які включають різні автономні інструменти та менеджери для спрощення керування, моніторингу та автоматизації складного підприємства. Ці рішення базуються на функціональній декомпозиції управління, що виконується декількома кооперативними менеджерами з різними цілями управління (наприклад, менеджер продуктивності, менеджер живлення, менеджер зберігання тощо). Ці інструменти спеціально розроблені та оптимізовані для використання в IT-інфраструктурі підприємств і центрів обробки даних.

Самоуправління може бути централізованим, децентралізованим або гібридним (ієрархічним). Більшість підходів до самоуправління засновані або на централізованому управлінні, або припускають високу доступність

макромасштабної, точної та актуальної інформації про керовану систему та середовище її виконання. Останнє припущення є нереалістичним для високодинамічних великомасштабних розподілених систем із кількома власниками, наприклад систем P2P, спільнотних мереж і хмар. Як правило, самокерування в інформаційній системі підприємства, мережі доставки вмісту (CDN - Content Delivery Network) від одного постачальника або хмарі центру обробки даних є централізованим, оскільки більшість управлінських рішень приймається на основі глобального (макромасштабного) стану системи, щоб досягти близької до оптимальної роботи системи. Однак централізоване керування не є масштабованим і може бути ненадійним.

Існує багато проектів, у яких використовуються такі методи, як теорія управління, машинне навчання, емпіричне моделювання або їх комбінація для досягнення SLO на різних рівнях багаторівневої програми Web 2.0.

Наприклад, Lim et al. [43] запропонували використовувати два контролери для автоматизації еластичності сховища. Інтегрований контролер зворотного зв'язку використовується для підтримки середнього часу відгуку на бажаному рівні. Оптимізація на основі витрат використовується для контролю впливу операції ребалансування, необхідної для зміни розміру еластичного сховища, на час відгуку. Автори також пропонують використовувати пропорційне порогове значення, техніку, необхідну для уникнення коливань при роботі з дискретними системами. Конструкція контролера зі зворотним зв’язком базується на високій кореляції між використанням центрального процесора та середнім часом відгуку.

Сфера автономних обчислень все ще розвивається. Існує багато відкритих дослідницьких питань, таких як середовища розробки для сприяння розробці програм із самокеруванням, ефективний моніторинг, масштабована активація та надійне керування. Наша робота вносить внесок у сучасний рівень автономних обчислень, зокрема, самокерування великомасштабними та/або динамічними розподіленими системами. Ми вирішуємо кілька проблем, таких як автоматизація контролю еластичності,

надійність управління, розподіл функціональних можливостей управління між кооперативними автономними менеджерами та програмування самокерованих програм.

2.2 Опис підходу до побудови розподіленої системи керування компонентами

Автономні обчислення [5] є привабливою парадигмою для вирішення проблеми зростаючої складності програмного забезпечення, роблячи програмні системи та програми самокерованими. Самоуправління, а саме самоконфігурація, самооптимізація, самовідновлення та самозахист можуть бути досягнуті за допомогою автономних менеджерів [6]. Автономний менеджер постійно контролює програмне забезпечення та середовище його виконання та діє для досягнення цілей управління. Управління програмами в динамічних середовищах з динамічними ресурсами та/або навантаженням (наприклад, Grids, однорангові системи та хмари) є особливо складним через великий масштаб, складність, високий відтік ресурсів (наприклад, у системах P2P) і відсутність чіткої відповідальності керівництва.

Розглянемо підхід до розробки платформи для самокерованих розподілених програм. Система керування розподіленими компонентами загального призначення, яка використовується для розробки, розгортання та виконання самокерованих розподілених програм або служб у різних середовищах, у тому числі дуже динамічних із нестабільними ресурсами. Дана система — це як компонентна модель програмування, яка включає аспекти керування, так і розподілене середовище виконання.

Система надає середовище програмування, яке спеціально розроблене, щоб дозволити розробникам програм проектувати та розробляти складні розподілені програми, які запускатимуться та керуватимуть самостійно в динамічних та нестабільних середовищах. Нестійкість може бути пов’язана з ресурсами (наприклад, ресурсами нижчого рівня), змінним навантаженням

або діями інших програм, що працюють на тій самій інфраструктурі. Бачення полягає в тому, що після встановлення середовища виконання для всієї інфраструктури програми, розроблені за її допомогою, можна буде інсталювати та запускати практично без зусиль. Після розгортання додаток керує собою, абсолютно без втручання людини, за винятком, звичайно, змін політики. Протягом життя програми програма прозоро відновлюється після збою, налаштовується та переконфігурується відповідно до змін навколишнього середовища, таких як доступність ресурсів або навантаження. Сьогодні це неможливо зробити в нестабільних середовищах, тобто це виходить за рамки сучасного рівня техніки, за винятком прикладних програм на одній машині та найбільш тривіальних розподілених програм, наприклад, клієнт/сервер.

Переваги самокерованих програм застосовуються в усіх середовищах, а не лише в динамічних. Альтернативою самоуправлінню є управління людьми, яке є дорогим, схильним до помилок і повільним. У відомій ініціативі IBM Autonomic Computing Initiative [5] осями самоуправління були самоконфігурація, самовідновлення, самоналаштування та самозахист. На сьогоднішній день у цій сфері ведеться значна робота, більша частина якої спрямована на кластери.

Однак чим динамічніше та нестабільніше середовище, тим частіше будуть потрібні відповідні дії керування для відновлення/налаштування/ переналаштування програми. У дуже динамічному середовищі самокерування – це не питання вартості, а питання здійсненності, оскільки управління людьми (навіть якщо їх можна було б зібрати достатньо) буде надто повільним, і система деградуватиме швидше, ніж люди зможуть її відновити. Будь-яка нетривіальна розподілена програма, що працює в такому середовищі, повинна бути самокерованою . Є кілька розподілених програм, які є самокерованими та можуть працювати в динамічних середовищах, як-от однорангові системи обміну файлами, але вони створені вручну та призначені

для спеціального призначення, не пропонують вказівок щодо розробки самокерованих розподілених програм у загальний.

Управління програмами в розподіленому налаштуванні складається з двох частин. По-перше, це початкове розгортання та конфігурація, коли окремі компоненти доставляються, розгортаються та ініціалізуються на відповідних вузлах (або екземплярах віртуальної машини), потім компоненти зв'язуються один з одним відповідно до архітектури програми, і програма може початок роботи. По-друге, існує динамічна реконфігурація, коли запущену програму потрібно переконфігурувати. Зазвичай це пов'язано зі змінами навколишнього середовища, такими як зміна навантаження, стан інших програм, які спільно використовують ту саму інфраструктуру, збій вузла, вихід вузла (або власник скасовує спільне використання свого ресурсу, або контрольоване завершення роботи), але це також може бути наслідком до помилок програмного забезпечення або змін політики. Усі завдання в початковій конфігурації також можуть бути присутніми в динамічній реконфігурації. Наприклад, збільшення кількості вузлів на певному рівні включатиме виявлення відповідних ресурсів, розгортання та ініціалізацію компонентів на цих ресурсах і відповідне їх прив'язування. Однак динамічна реконфігурація, як правило, передбачає більше, тому що, по-перше, програма запущена, і збої повинні бути зведені до мінімуму, а по-друге, керівництво повинно мати можливість маніпулювати запущеними компонентами та існуючими прив'язками. Загалом , у динамічній реконфігурації існує більше обмежень щодо порядку виконання дій зі зміни конфігурації порівняно з початковою конфігурацією, коли конфігурація може бути створена спочатку, а компоненти активуються лише після її завершення.

Конфігурацію можна розглядати як граф, де вузли є компонентами, а посилання є прив'язками. Компонентам потрібні відповідні ресурси для їх розміщення, і ми можемо завершити картину, додавши відображення компонентів на фізичні ресурси. Це показано на рисунку 2.2. Ліворуч ми показуємо лише графік, абстрактну конфігурацію, тоді як праворуч показана

конкретна конфігурація. Прив’язки, які перетинають межі ресурсів, під час використання включатимуть віддалені виклики, тоді як ті, які цього не роблять, можуть бути викликані локально. Реконфігурація може передбачати зміну лише конкретної конфігурації або і абстрактної, і конкретної конфігурацій. Зверніть увагу, що ми показуємо більш цікаві та складні аспекти реконфігурації; є також реконфігурації, які залишають графік без змін, але лише змінюють спосіб роботи компонентів шляхом зміни атрибутів компонентів.

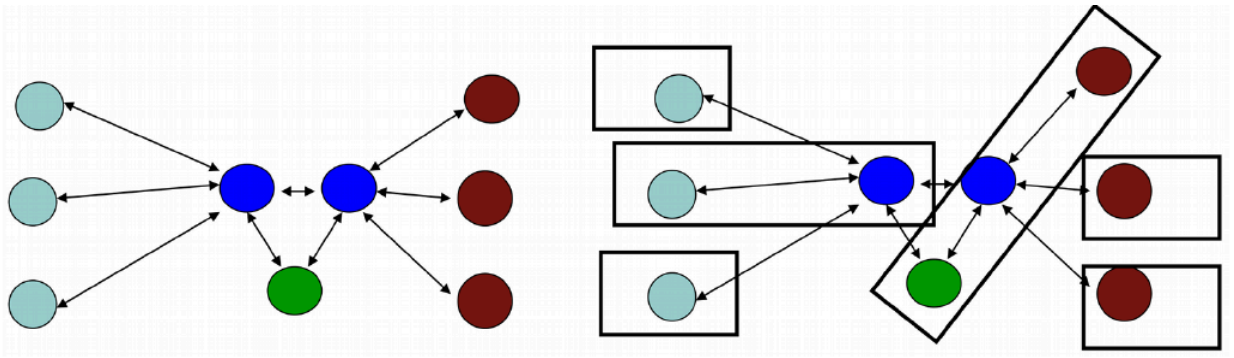


Рисунок 2.2 – Абстрактний (ліворуч) і конкретний (праворуч) вигляд конфігурації. Коробки представляють вузли або віртуальні машини, кола представляють компоненти

Тепер ми перейдемо до деяких прикладів динамічної реконфігурації. У цих динамічних середовищах ресурс може оголосити, що він покидає, і потрібно буде знайти новий ресурс, а компоненти, які зараз знаходяться на ресурсі, перемістити до нового ресурсу. У цьому випадку змінюється лише конфігурація основи. Крім того, коли кількість компонентів служби на рівні служби збільшується, це змінить абстрактну (і конкретну) конфігурацію шляхом додавання нового вузла та відповідних прив’язок. Іншим прикладом є збій ресурсу. Якщо ми знехтуємо тимчасовою пошкодженою конфігурацією, де несправний компонент більше не присутній у конфігурації, а прив’язки, які існували до нього, порушені, зрештою буде створено ідентичну абстрактну конфігурацію, що відрізняється лише відображенням ресурсу.

Загалом, архітектура програми складається з набору відповідних абстрактних конфігурацій із пов’язаною інформацією щодо вимог до ресурсів компонентів. Фактичне середовище визначатиме, яке з них найкраще розгорнути або переналаштувати.

На рисунку 2.1 показано лише компоненти верхнього рівня. На вищому рівні деталізації існує набагато більше компонентів, але для нашого керування ми можемо ігнорувати компоненти, які завжди розташовані разом і прив’язані виключно до компонентів, розташованих разом. Зауважте, що ми ігноруємо лише ті, які завжди розташовані разом (у всіх конфігураціях). Існують компоненти, які можуть бути розташовані в деяких конкретних конфігураціях (якщо доступний достатній потужний ресурс), але не в інших. На рисунку 2.1 праворуч показана конфігурація з однією машиною, що містить 3 компоненти; в іншій конкретній конфігурації вони можуть бути зіставлені з іншими машинами.

Ми використовуємо архітектурний підхід до самоуправління, приділяючи особливу увагу на досягнення самоуправління для динамічних середовищ, уможливлюючи використання кількох розподілених кооперативних автономних менеджерів для масштабованості.

2.3 Розробка структури системи самоналаштовуваного адаптивного керування обчислювальною системою

Розглянемо структуру платформи для розробки, розгортання та виконання самокерованих програм на основі компонентів яка включає розподілену модель програмування компонентів, API та систему виконання (включаючи службу розгортання), яка працює у внутрішній структурованій накладеній мережі. Система підтримує визначення змін у стані компонентів і середовищі виконання, а також дозволяє знаходити окремі компоненти та відповідним чином керувати ними. Він розгортає як функціональні

компоненти, так і компоненти керування, а також налаштовує відповідну інфраструктуру підтримки датчиків і активації.

Пропоновану систему розроблено з припущенням, що його середовище виконання та додатки можуть виконуватися в дуже динамічному середовищі з нестабільними ресурсами, де ресурси (комп'ютери, віртуальні машини) можуть непередбачувано вийти з ладу або залишитися. Щоб впоратися з такою динамічністю, використовуються властивості самоорганізації основної структурованої накладної мережі, включаючи маршрутизацію на основі імен і функціональність DHT. Для ефективності система безпосередньо підтримує абстракцію групи компонентів із груповими прив'язками (один до всіх і один до всіх).

Є аспекти, які досить поширені в автономних обчисленнях. По-перше, пропонована система ідтримує парадигму циклу управління зі зворотним зв'язком, де логіка управління в безперервному циклі зворотного зв'язку відчуває зміни в середовищі та статусі компонентів, причини цих змін, а потім, коли це необхідно, активує, тобто маніпулює компонентами та їх прив'язками. Додаток із самокеруванням можна розділити на функціональну частину та частину керування, пов'язану між собою за допомогою датчиків і активації. По-друге, модель нішевого програмування базується на моделі компонентів, яка називається фрактальною моделлю компонентів [33], у якій компоненти можна контролювати та керувати ними. Фрактальні компоненти пов'язані та функціонально взаємодіють один з одним за допомогою двох видів інтерфейсів: (1) інтерфейси сервера, які пропонують компоненти; (2) клієнтські інтерфейси, які використовуються компонентами. Компоненти -взаємопов'язані прив'язками: клієнтський інтерфейс одного компонента прив'язаний до серверного інтерфейсу іншого компонента. Fractal дозволяє вкладати компоненти в складені компоненти та спільно використовувати компоненти. Компоненти мають мембрани контролю (керування) з можливостями самоаналізу та заступництва. Саме через цю керуючу мембрану компоненти запускаються, зупиняються, налаштовуються. Саме

через цю мембрану відбувається пасивація компонентів, через яку компонент може повідомляти про події, що стосуються програми (наприклад, навантаження). Фрактал можна розглядати як визначення набору можливостей для функціональних компонентів. Це не вимагає відповідності компонентів додатків , але чітко можливості запрограмованих компонентів повинні відповідати потребам управління. Наприклад, якщо компонент має статус і не здатний до пасивації (або контрольних точок), тоді керівництво не зможе прозоро перемістити компонент.

Головною новою особливістю системи є те, що для того, щоб увімкнути та досягти самоуправління для великомасштабних динамічних розподілених систем, вона поєднує відповідну модель компонентів (Fractal) із структурованою накладеною мережею, подібною до Chord, для надання ряду надійних накладених служб. Пропонована система використовує властивості самоорганізації структурованої накладної мережі, наприклад, автоматичне виправлення таблиць маршрутизації на вихідних вузлах, з'єднаннях і збоях. Модель Fractal підтримує компоненти, які можна відстежувати та керувати за допомогою інтерфейсів самоаналізу та контролю компонентів (так звані контролери у Fractal), наприклад контролери життєвого циклу, атрибутів, прив'язки та контенту. Середовище виконання надає ряд накладених служб, зокрема зв'язок на основі імен, сховище ключ-значення (DHT) для служб пошуку, контрольовану трансляцію для виявлення ресурсів, механізм публікації/підписки для розповсюдження подій та вузол виявлення збоїв. Ці служби використовуються для надання абстракцій вищого рівня, таких як прив'язки на основі імен для підтримки мобільності компонентів; групи динамічних компонентів; групові прив'язки «один до будь-кого» та «один до всіх» і взаємодія на основі подій. Зауважте, що програмісту додатків не потрібно знати про базові служби накладання, це під капотом, і його/її взаємодія здійснюється через API.

Тепер ми переходимо до опису середовища виконання і меншою мірою, моделі програмування.

Пропонована система реалізує еталонну архітектуру автономних обчислень, запропоновану IBM у [6], тобто дозволяє створювати цикли керування MAPE-K (Monitor, Analyze, Plan and Execute with Knowledge). Автономний менеджер може бути організований як мережа елементів керування (ME), які взаємодіють через події, контролюють через датчики та діють через виконавчі механізми (наприклад, за допомогою API активації). Можливість розподіляти ME між контейнерами дає змогу побудувати децентралізовані контури керування зі зворотним зв’язком для надійності та продуктивності.

Самокерована програма складається з функціональної та керуючої частин. Функціональні компоненти спілкуються через прив’язки компонентів, які зв’язують інтерфейси клієнта з інтерфейсами сервера; тоді як елементи керування спілкуються переважно через механізм сповіщення про події публікації/підписки. Функціональна частина розроблена з використанням фрактальних компонентів і груп компонентів, які є контрольними мітками (наприклад, їх можна шукати вгору, переміщувати, відбивати, запускати, зупиняти тощо) і контролювати за допомогою керуючої частини програми. Частина керування програмою може бути побудована як набір інтерактивних або незалежних циклів керування, кожна з яких контролює певну частину програми та реагує на заздалегідь визначені події, такі як збої вузлів, вихід або приєднання, збої компонентів і події членства в групах; і події, пов’язані з програмою, такі як події зміни навантаження компонентів і події низької ємності пам’яті.

На рисунку 2.3 ми показуємо, як може виглядати абстрактна конфігурація, коли всі елементи керування пасивні в тому сенсі, що всі вони очікують на виконання певних ініціюючих подій. Двокінцеві стрілки у функціональній частині є прив’язками між компонентами (оскільки конкретна конфігурація не показана, прив’язки можуть бути або не бути між різними машинами). Елементи керування мають посилання на функціональні компоненти за назвою (наприклад, ідентифікатор компонента) або підключені

до приводів. Управлінська та функціональна частини також «підключені» за датчиками (це також фактично за назвою, оскільки керування, а також функціональні компоненти можуть мігрувати) На зображенні датчики з групи А функціональних компонентів (A1, A2 і A3) до двох елементів керування (датчики, підключені до інші елементи керування не показані).

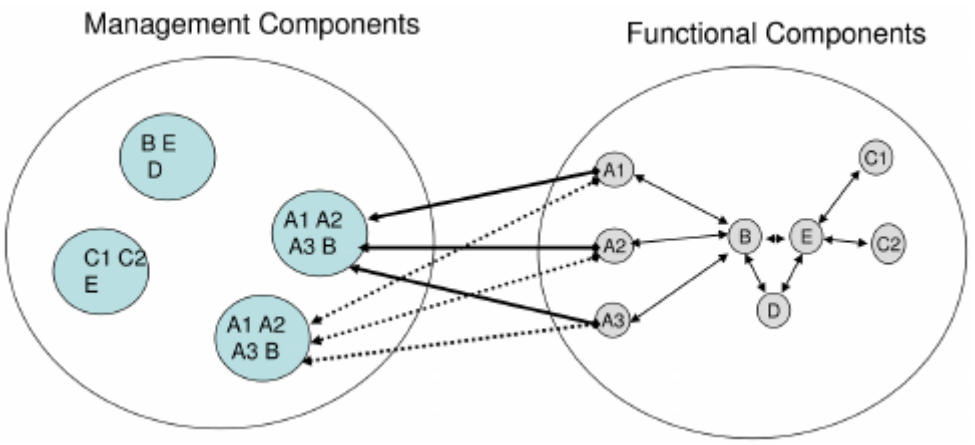


Рисунок 2.3 – Абстрактна конфігурація самокерованої програми

Архітектура керування на рисунку 2.3 є плоскою, а пізніше ми покажемо, як управління можна структурувати ієрархічно, що важливо для великих і складних програм.

Нижче наведено форму елемента керування разом із описом високого рівня функцій, доступних у API активації системи:

```
loop
     wait SensorEvent
        change internal state   // e.g., for monitoring and aggregation
        analyze/plan
        actuate
```

Активація – це послідовність викликів (дій), які перераховані нижче (без певного порядку). Зауважте, що всі наведені нижче дії надаються в API активації. Список розширюється за допомогою визначених користувачем дій:

```
reconfigure existing components    // functional components
                                   //changing concrete configuration only
passivate/move existing components
discover resources     // functional components / changing  configuration.
allocate and deploy new components on a given resource
kill/remove existing components
remove/create bindings
add subscriptions/sensors  // may cause sensors to be installed
remove subscriptions


discover resources                        // management components
allocate resources and deploy new management elements
trigger events  // for management coordination
```

Для впровадження точок дотику (датчиків і виконавчих механізмів) пропонована система використовує функції самоаналізу та динамічної реконфігурації компонентної моделі Fractal, щоб забезпечити абстракції API датчиків і активації. Датчики та виконавчі механізми — це спеціальні компоненти, які можна приєднати до функціональних компонентів програми. В системі також є вбудовані датчики, які виявляють зміни в середовищі, такі як збої ресурсів і компонентів, приєднання та вихід, а також зміни в архітектурі програми, такі як створення групи.

Прикладному програмісту також необхідно встановити/розгорнути елементи (компоненти) керування. Значною мірою це робиться аналогічно роботі з функціональними компонентами. Однак є дві важливі відмінності. Один стосується розподілу ресурсів для компонентів керування хостом, а інший стосується зв’язків між елементами керування. В системі прикладний програміст зазвичай дозволяє середовищу виконання знайти відповідний ресурс і розгорнути компонент керування за один крок. Система резервує частину кожної машини для управлінської діяльності, щоб елементи керування можна було розмістити де завгодно (в ідеалі, оптимально, щоб мінімізувати затримку між елементом керування та його датчиками та посиланнями). Зауважте, що це припускає, що крок аналізу/планування в логіці керування є недорогим обчислювальним. По-друге, існують інші способи явного спільного використання елементів управління, і вони рідко

пов’язані один з одним (якщо вони не завжди розташовані разом). На рисунку 2.3 відсутні будь-які зв’язки між елементами управління, тому єдина координація, яка можлива між менеджерами – це стигмергія. Знаннями (як у MAPE-K) можна ділитися між ME за допомогою двох механізмів: по-перше, механізм публікації/підписки; по-друге DHT для зберігання/вилучення інформації, такої як посилання на членів групи компонентів, відображення імені та розташування.

Наразі в пропонованій системі така підтримка мов високого рівня включає декларативний ADL (мова опису архітектури), який використовується для опису початкових конфігурацій на високому рівні, який інтерпретується під час виконання для початкового розгортання.

2.4 Архітектура пропонованої системи керування

Середовище виконання (рис. 2.4) — це набір розподілених контейнерів, створених для мережі і ряду служб, включаючи зв’язок на основі імен, виявлення ресурсів, розгортання, служб пошуку, підтримки компонентів включаючи сповіщення про попередньо визначені події (наприклад, збої компонентів).

Служби дозволяють програмі виявляти та розподіляти ресурси, розгортати програму та переконфігурувати її під час виконання, відстежувати та реагувати на зміни в програмі та середовищі її виконання, а також знаходити елементи застосування (наприклад, компоненти, групи, менеджери). У цьому підрозділі ми опишемо середовище виконання. Ми починаємо з аспектів середовища виконання, про які повинен знати прикладний програміст. Після цього ми опишемо механізми, які використовуються для реалізації середовища виконання, і, зокрема, накладених служб. Хоча прикладному програмісту не потрібно розуміти основні механізми, вони відображені в моделі продуктивності/помилок. Також в в цьому розділі ми описуємо модель «продуктивність/помилка».

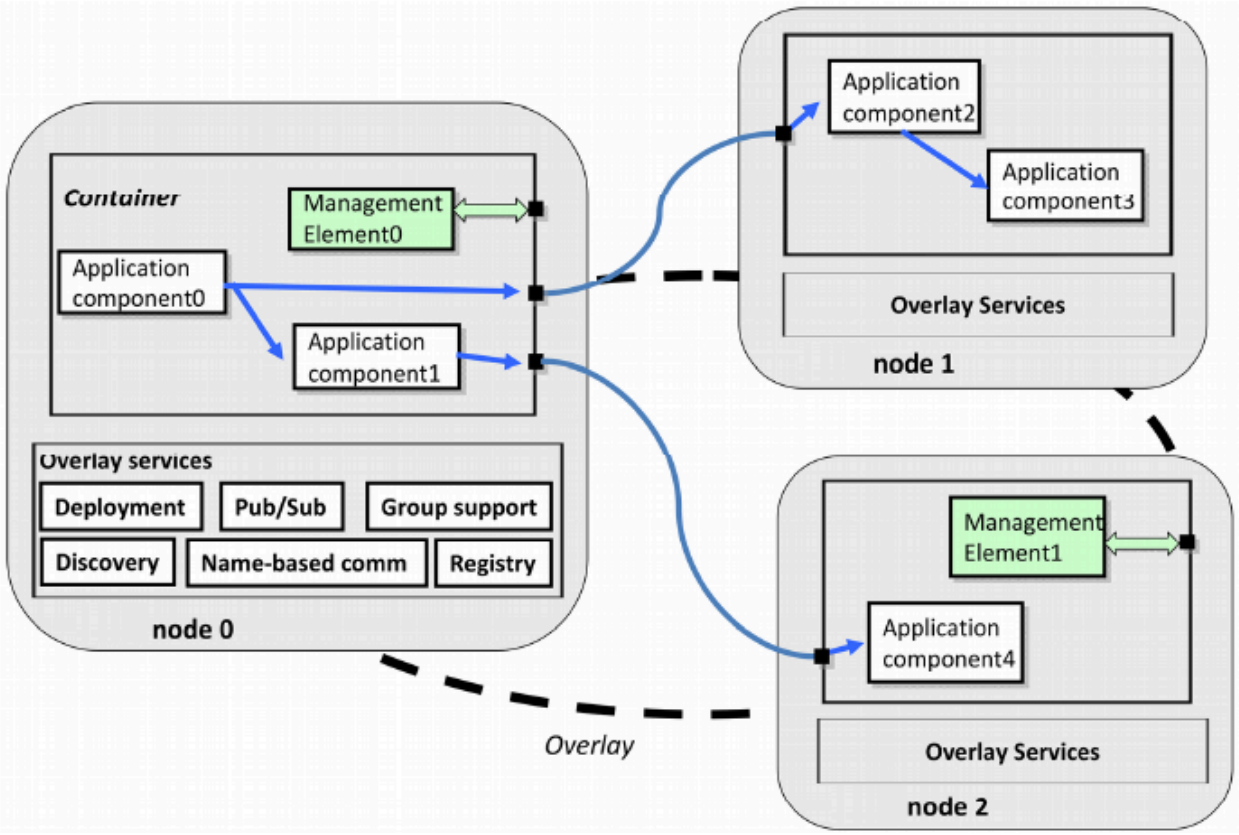


Рисунок 2.4 – Архітектура пропонованої системи

Середовище виконання адаптивної системи — це набір розподілених контейнерів, які називаються вузлами Jade, з'єднаних через структуровану P2P-мережу. Контейнери містять функціональні компоненти та елементи керування розподілених програм, які виконуються в ніші. У поточному прототипі є дві конфігурації контейнера:

1. контейнер JadeBoot, який завантажує систему та інтерпретує задані файли ADL (*.fractal), що описують початкову конфігурацію програми під час розгортання;

2. контейнер JadeNode, який не містить ADL в інтерпретаторі, але підтримує API розгортання для програмного розгортання компонентів.

Ми використовуємо програму Webcache PHP (розгорнуту на сервері Apache) для підтримки списку вузлів, які використовуються як точки доступу для приєднання до накладеної мережі. URL-адреса Webcache є частиною конфігураційної інформації, яка надається під час встановлення та налаштування платформи. Після запуску новий вузол Jade надсилає

HTTP-запит до Webcache, щоб отримати адресу будь-якого з вузлів Jade, до якого можна зв’язатися, щоб приєднатися до накладання.

Адаптивна система дозволяє програмісту контролювати розподіл функціональних компонентів і елементів керування серед контейнерів, тобто для кожного компонента ME програміст може вказати контейнер (за ідентифікатором ресурсу), де цей елемент має розташовуватися. Якщо розташування не вказано, служба розгортання середовища виконання розгорне (або перемістить у разі помилки) ME на будь-якому контейнері, вибраному випадковим чином або циклічним способом. Розміщення ME з контрольованим компонентом в одному контейнері дозволяє покращити ефективність керування шляхом моніторингу та/або керування компонентом локально, а не віддалено через мережу.

Система забезпечує підтримку груп компонентів і групових зв’язків. Компоненти можна прив’язувати до груп через прив’язки «один до будь-кого» (де член групи вибирається випадковим чином) або «один до всіх». Використання груп компонентів є досить поширеним шаблоном програмування. Наприклад, рівень у багаторівневій програмі може бути змодельований як група компонентів. Програміст прикладної програми повинен знати про те, що групи компонентів підтримуються безпосередньо під час виконання з міркувань ефективності (альтернативою було б запрограмувати групову абстракцію).

Дана система – це інфраструктура, яка вільно з’єднує доступні фізичні ресурси/контейнери (комп’ютери) і забезпечує виявлення ресурсів. Середовище виконання — це набір контейнерів (компонентів і менеджерів хостингу), які після приєднання та виходу з оверлею інформують середовище виконання та його додатки у спосіб, повністю аналогічний одноранговим системам (наприклад, Chord).

Для початкового розгортання та реконфігурації під час виконання система надає послугу розгортання (включно з виявленням ресурсів), яку може виконати інтерпретатор ADL, надавши ADL (можливо, неповний) опис

архітектури програми, що розгортається; або програмно за допомогою розгортання API. Розгортання програми, кероване ADL, не обов’язково розгортає всю програму, а скоріше деякі основні компоненти, які, у свою чергу, можуть завершити розгортання програмним шляхом, виконавши логіку процесу розгортання. Процес розгортання включає виявлення ресурсів, розміщення та створення компонентів і груп компонентів, прив’язування компонентів і груп, розміщення та створення елементів керування, підписку на попередньо визначені події або події, пов’язані з програмою. Служба розгортання (API) використовує службу виявлення ресурсів ніші для пошуку ресурсів (контейнерів ніші) із зазначеними властивостями для розгортання компонентів.

Усе заплановане видалення ресурсів, як-от контрольоване завершення роботи, має здійснюватися шляхом виконання дії виходу незадовго до видалення ресурсу. Керівництву, як правило, легше виконати необхідну реконфігурацію на листках, ніж на збоях. Сподіваємося, керівництво мало час, необхідний для успішного переміщення (або знищення) компонентів, розміщених на ресурсі, до моменту фактичного видалення ресурсу з інфраструктури (наприклад, вимкнення).

На додаток до служб виявлення ресурсів і розгортання, описаних вище, підтримка системи виконання для самостійного керування включає службу публікації/підписки, яка використовується для моніторингу та керування подіями; і ряд серверних інтерфейсів для керування компонентами, групами та елементами керування, а також для доступу до накладених служб (виявлення, розгортання та pub/sub).

Послуга публікації використовується елементами керування для публікації та доставки подій моніторингу та активації. Доступ до служби здійснюється через системні інтерфейси ActuatorInterface і TriggerInterface, описані нижче. Служба надає вбудовані датчики для моніторингу збоїв компонентів і вузлів/залишення та зміни членства в групах. Датчики видають відповідні попередньо визначені події (наприклад, ComponentFailEvent,

CreateGroupEvent, MemberAddedEvent, ResourceJoinEvent, ResourceLeaveEvent, ResourceStateChangeEvent), на які ME можуть підписатися. Відповідний API pub/sub дозволяє програмісту також визначати датчики та події для певної програми. Система Niche runtime гарантує доставку подій.

Система виконання надає ряд інтерфейсів (доступних у кожному контейнері), які використовуються ME для керування функціональною частиною програми та доступу до накладених служб (виявлення, розгортання, pub/sub). Інтерфейси автоматично прив'язуються системою виконання до відповідних клієнтських інтерфейсів ME, коли елемент керування розгортається та ініціалізується. Набір інтерфейсів середовища виконання включає такі інтерфейси:

- ActuatorInterface надає методи для доступу до непрофесійних служб, для (від)зв'язування функціональних компонентів, для маніпулювання групами, для отримання доступу до компонентів для моніторингу та контролю за ними (тобто для реєстрації компонентів і ME з іменами та для пошуку за іменами). Методи цього інтерфейсу включають, але не обмежуються цим, виявлення, виділення, звільнення, розгортання, повторне розгортання, підписка, скасування підписки, реєстрація, пошук, прив'язування, розв'язування, створення групи, видалення групи, додавання до групи;
- TriggerInterface, який використовується для запуску подій;
- Registry — це допоміжний низькорівневий інтерфейс, який використовується для пошуку компонентів за загальносистемними іменами;
- OverlayAccess — це допоміжний низькорівневий інтерфейс, який використовується для отримання доступу до системи виконання та інтерфейсу ActuatorInterface.

При розробці частини програми для керування розробник повинен переважно використовувати перші два інтерфейси. Зауважте, що на додаток до вищезазначених інтерфейсів програміст також використовує API

компонентів і груп (Fractal API) для маніпулювання компонентами та групами з метою самостійного керування. Архітектурні елементи (компоненти, групи, ME) можуть розташовуватися в різних нішових контейнерах; тому виклики методів інтерфейсу ActuatorInterface, а також групових і компонентних інтерфейсів можуть бути віддаленими, тобто перетинати межі контейнера. Усі архітектурні елементи (компоненти, групи, елементи керування) програми унікально ідентифікуються за допомогою загальносистемних ідентифікаторів, призначених під час розгортання. Елемент можна зареєструвати в системі середовища виконання із заданим іменем, щоб шукати (і зв'язувати з ним) його ім'я.

На рисунку 2.5 зображено етапи виконання (віддаленого) виклику методу для компонента, розташованого у віддаленому контейнері.

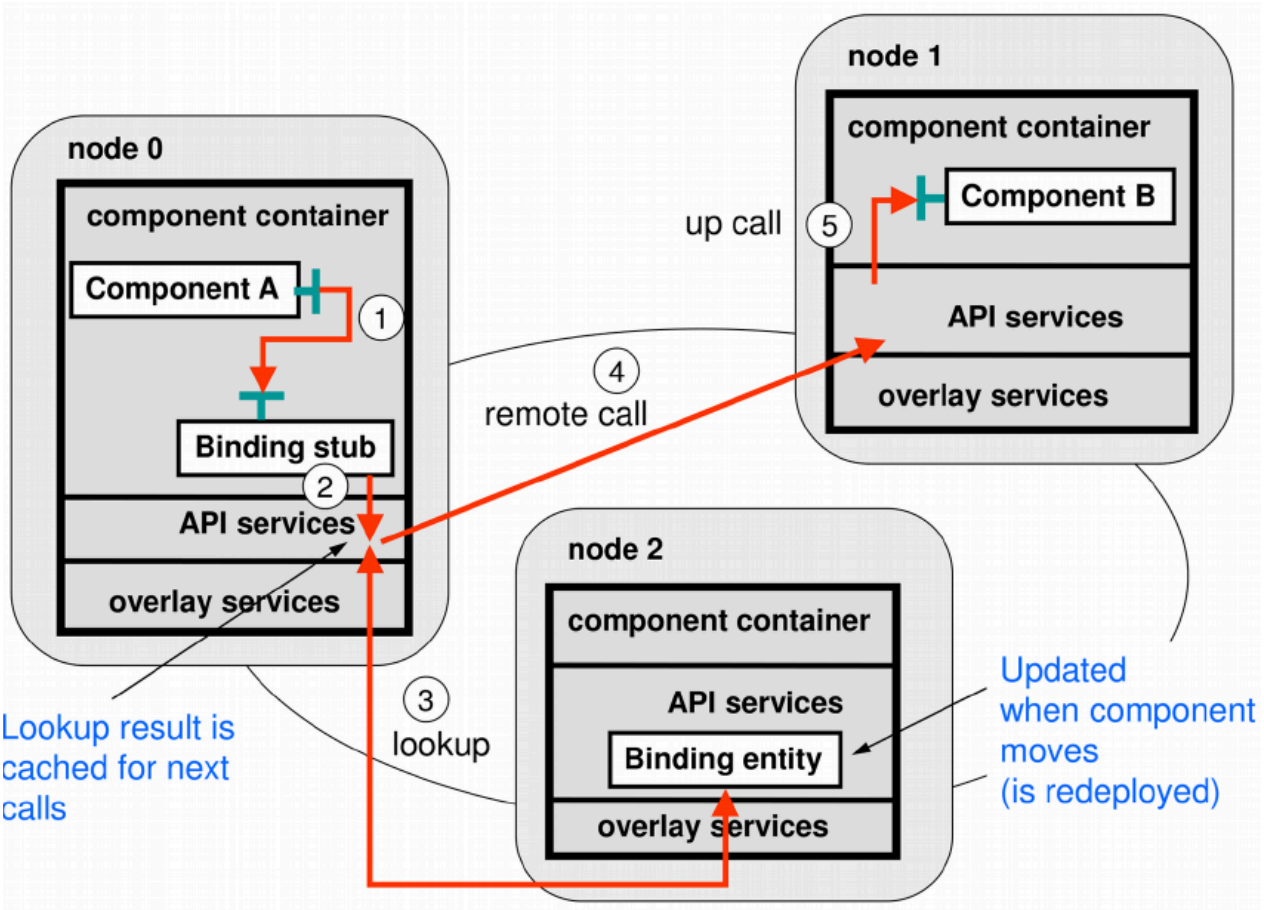


Рисунок 2.5 – Етапи виклику методу в системі

Припустимо, що клієнтський інтерфейс компонента A у вузлі 0 прив’язаний до серверного інтерфейсу компонента B у вузлі 1; тоді як інформація про прив’язку A до B (тобто кінцева точка B) зберігається у вузлі 2. Коли A здійснює свій перший виклик до B (Крок 1), виклик методу викликається на заглушці прив’язки B у вузлі 0 (Крок 2). Заглушка виконує пошук, використовуючи ідентифікатор прив’язки як ключ, для поточного розташування компонента B (Крок 3). Результат пошуку, тобто посилання на кінцеву точку B, кешується у вузлі 0 для подальших викликів. Коли посилання на B вирішено, заглушка здійснює віддалений виклик до компонента B, використовуючи посилання. Усі подальші виклики B з вузла 0 використовуватимуть кешоване посилання на кінцеву точку. Якщо з будь-якої причини B переходить до іншого контейнера (не показано на рисунку 2.5) , розташування B буде оновлено в DHT, і заглушка B у вузлі 0 зможе знайти нове розташування під час наступного виклику компонента B Якщо вузол, на якому розміщено компонент B, виходить з ладу, повідомлення про збій компонента буде надіслано всім підписникам, включаючи менеджера (якщо є), відповідального за відновлення компонента B в іншому контейнері. У цьому випадку компонент A, який прив'язаний до B, не потребує інформування; повторне прив’язування A до нового екземпляра B виконується прозоро для A.

Інформація про місцезнаходження зберігається в DHT у формі структури даних під назвою Set of Network References, SNR, яка представляє набір посилань на ідентифіковані елементи (наприклад, компоненти, групи компонентів). SNR компонента містить одне посилання, тоді як SNR групи компонентів містить посилання на членів відповідної групи. SNR зберігаються під їхніми іменами (використовуються як ключі) у сховищі ключ-значення на основі DHT. SNR використовуються для пошуку елементів ніші за іменами та можуть містити прямі або непрямі посилання. Пряме посилання містить розташування елемента; тоді як непряме посилання посилається на інший SNR, визначений його назвою. Непряме посилання має

бути вирішено перед використанням. Клієнт може кешувати SNR, щоб покращити час доступу до елемента, на який посилаються. Система прозоро виявляє застарілі (недійсні) посилання та за потреби оновлює вміст кешу. Система підтримує прозоре визначення елементів, на які посилається SNR. Коли елемент керування створюється для керування (відчуття та приведення в дію) функціональних компонентів, на які посилається SNR, система виконання Система розгортає датчики та виконавчі механізми для кожного компонента. Щоразу, коли посилання в SNR змінюються, система виконання розгортає датчики та виконавчі механізми для відповідних компонентів. Для надійності SNR реплікуються за допомогою механізму реплікації DHT. Реплікація SRN забезпечує можливу узгодженість реплік SNR, але дозволені тимчасові узгодженості . Подібно до обробки кешування SNR, структура розпізнає застарілі посилання на SNR і повторює спробу доступу до SNR, коли це необхідно.

Самокерування вимагає моніторингу середовища виконання, компонентів і груп компонентів. В даному випадку моніторинг виконується методом push, а не pull для підвищення продуктивності та масштабованості за допомогою механізму розповсюдження подій публікації/підписки. Датчики та елементи керування можуть публікувати заздалегідь визначені (наприклад, збій вузла) і специфічні для програми (наприклад, зміна навантаження) події, які будуть доставлені передплатникам (слухачам подій).

Пропонована система дозволяє максимально децентралізувати управління. Управління можна розділити (тобто розпаралелювати) за аспектами (наприклад, самовідновлення, самоналаштування), просторово та ієрархічно. На нашу думку, одна програма має багато слабко синхронізованих менеджерів. Система підтримує мобільність елементів управління і також забезпечує платформу виконання для цих менеджерів; вони зазвичай призначаються різним машинам у накладенні. Існує певна підтримка для оптимізації цього розміщення менеджерів і певна підтримка реплікації менеджерів для відмовостійкості. Таким чином, вирішується, принаймні

частково, завдання уникнути вузького місця в управлінні. Основна причина «принаймні частково» в останньому реченні полягає в тому, що, ймовірно, буде потрібна додаткова підтримка оптимального розміщення менеджерів, враховуючи локальність мережі.

Висновки до розділу 2

В даному розділі представлена структура управління яка дозволяє розробляти розподілені компонентні додатки із самоналаштовуваною системою керування, які не залежать від функціонального коду додатка, але можуть взаємодіяти з ним, коли це необхідно. Фреймворк забезпечує невеликий набір абстракцій, які сприяють надійному й ефективному управлінню додатками навіть у динамічних середовищах. Фреймворк використовує властивості структурованої накладної мережі, на якій він побудований. В динамічних середовищах, таких як Grids або Clouds, самокерування представляє чотири проблеми. Пропонована архітектура здебільшого відповідає цим завданням і представляє модель програмування та службу виконання, щоб дозволити розробникам додатків розробляти самокеровані додатки.

РОЗДІЛ 3. ПОБУДОВА МОДЕЛІ ТА ШАБЛОНІВ ВЗАЄМОДІЇ РОЗПОДІЛЕНОГО МАСШТАБОВАНОГО ОБЧИСЛЮВАЛЬНОГО СЕРЕДОВИЩА

3.1 Представлення алгоритму розробки самоналаштовуваних додатків

Пропоноване адаптивне середовище дозволяє розробляти самокеровані програми, побудовані з функціональних компонентів і елементів керування. В цьому розділі ми докладніше описуємо модель програмування системи та наводимо приклад простої програми. Модель програмування базується на фрактальній, модульній і розширюваній компонентній моделі, призначеній для проектування, впровадження, розгортання та реконфігурації складних систем програмного забезпечення. Дана система запозичує основні концепції фракталів, якими є компоненти, інтерфейси та прив’язки, і додає нові концепції, пов’язані з груповим спілкуванням, розгортанням і керуванням.

3.1.1 Концепції адаптивного програмування

Самокерована програма складається з функціональних компонентів і елементів керування. Перші складають функціональну частину програми; тоді як останні складають частину управління.

Компоненти — це об’єкти середовища виконання, які взаємодіють виключно через іменовані чітко визначені точки доступу, які називаються інтерфейсами, включаючи інтерфейси керування, що використовуються для керування. Інтерфейси компонентів поділяються на два типи: клієнтські інтерфейси, які надсилають виклики операцій, і серверні інтерфейси, які їх отримують. Інтерфейси з’єднані через канали зв’язку, які називаються

прив’язками. Компоненти та інтерфейси називаються, щоб шукати інтерфейси компонентів за іменами та зв’язувати їх.

Компоненти можуть бути примітивними або складеними, утвореними шляхом ієрархічної збірки інших компонентів (так звані підкомпоненти). Ця ієрархічна композиція є ключовою функцією фракталів, яка допомагає керувати складністю розуміння та розробки компонентних систем.

Іншою важливою особливістю фракталу є його підтримка розширюваних відбиваючих засобів, що дозволяє перевіряти та адаптувати структуру та поведінку компонентів. Зокрема, кожен компонент складається з двох частин: мембрани, яка втілює рефлексивну поведінку та зміст, який складається з кінцевого набору підкомпонентів. Мембрана надає розширюваний набір інтерфейсів керування (так звані контролери) для реконфігурації внутрішніх функцій компонента та керування його життєвим циклом. Інтерфейси керування — це серверні інтерфейси, які мають бути реалізовані класами компонентів, щоб ними можна було керувати.

У пропонованій системі інтерфейси керування використовуються елементами керування, що стосуються конкретної програми (а саме, датчиками та приводами), а також середовищем виконання для моніторингу та керування компонентами, наприклад, для (повторного) прив’язування, зміни атрибутів і запуску.

Фрактал визначає наступні чотири основні інтерфейси керування: контролери атрибутів, зв’язування, вмісту та життєвого циклу. Контролер атрибутів (AttributeController) підтримує налаштування іменованих властивостей компонента. Контролер зв’язування (BindingController) використовується для зв’язування та від’єднання клієнтських інтерфейсів від інтерфейсів сервера, для пошуку інтерфейсу з заданим ім’ям і для переліку всіх клієнтських інтерфейсів компонента. Контролер вмісту (ContentController) підтримує перелік, додавання та видалення підкомпонентів. Нарешті, контролер життєвого циклу (LifeCycleController)

підтримує запуск і зупинку виконання компонента та отримання стану компонента.

Основні концепції фрактальної компонентної моделі представлено на рисунку 3.1, який зображує клієнт-серверну програму HelloWorld, яка є складеним фрактал компонентом, що містить два підкомпоненти, клієнт і сервер.

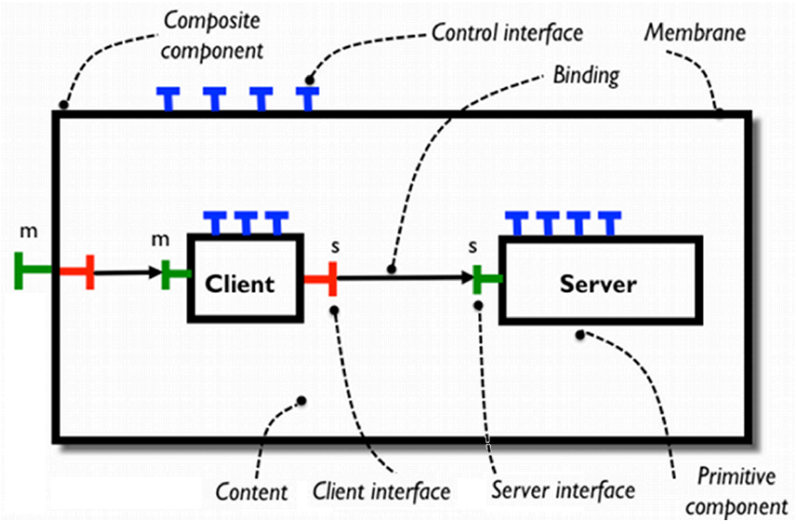


Рисунок 3.1 – Композитний фрактальний компонент HelloWorld із двома підкомпонентами клієнт і сервер

Клієнтський інтерфейс компонента «Клієнт» прив’язаний до інтерфейсу сервера компонента «Сервер». Мембрани компонентів містять інтерфейси керування. Зауважте, що під час розгортання композитний, клієнтський і серверний компоненти можна помістити в різні контейнери.

Створення програми на основі компонентів передбачає програмування примітивних компонентів і збирання їх у початкову конфігурацію програмним шляхом, використовуючи методи інтерфейсу ctuatorInterface середовища виконання або декларативно, використовуючи мову опису архітектури (ADL). У першому випадку принаймні один компонент (запуск) повинен бути описаний в ADL, щоб його початково розгортати та запускати інтерпретатор ADL. Компонент запуску може розгорнути частину програми,

що залишилася, виконавши робочий процес розгортання та налаштування, запрограмований за допомогою API активації середовища виконання, що дозволяє розробнику програмувати складні та гнучкі робочі процеси розгортання та налаштування. ADL, базується на Fractal ADL, розширюваній мові, що складається з модулів, кожен модуль визначає абстрактний синтаксис для певної архітектурної проблеми (наприклад, ієрархічне обмеження, розгортання). Примітивні компоненти програмуються на Java.

Пропонована система розширює компонентну модель абстракціями для групового зв'язку (група компонентів, прив'язування груп), а також абстракціями для розгортання та керування ресурсами (пакет, вузол).

Керівна частина програми програмується за допомогою абстракцій елементів керування (ME), які включають датчики, спостерігачі, агрегатори, менеджери, виконавці та виконавчі механізми. Зверніть увагу, що відмінність між спостерігачами, агрегаторами, менеджерами та виконавцями є архітектурною. З точки зору середовища виконання, усі вони є елементами керування, і керування можна запрограмувати однорідним способом (лише менеджери, датчики та виконавчі механізми). На рисунку 3.2 зображено типову ієрархію елементів управління в нішевому додатку . Ми розрізняємо різні типи ME залежно від ролі, яку вони відіграють у кодексі самоуправління. Датчики відстежують компоненти через інтерфейси та ініціюють події, щоб сповістити відповідні елементи керування про різні зміни, що стосуються окремих програм у контрольованих компонентах. Існують датчики, надані середовищем виконання для моніторингу збоїв/виходів компонентів (які, у свою чергу, можуть бути викликані збоями та виходами контейнерів/машин), груп компонентів (зміни членства, створення груп) і збоїв контейнерів. Спостерігачі отримують сповіщення про події від ряду датчиків, фільтрують і поширюють їх до агрегаторів, які збирають інформацію, виявляють симптоми та повідомляють менеджерам. Симптом є ознакою наявності певної аномалії у функціонуванні

контрольованих компонентів, груп або середовища. Керівники аналізують симптоми, приймають рішення та вимагають від виконавців діяти відповідно.

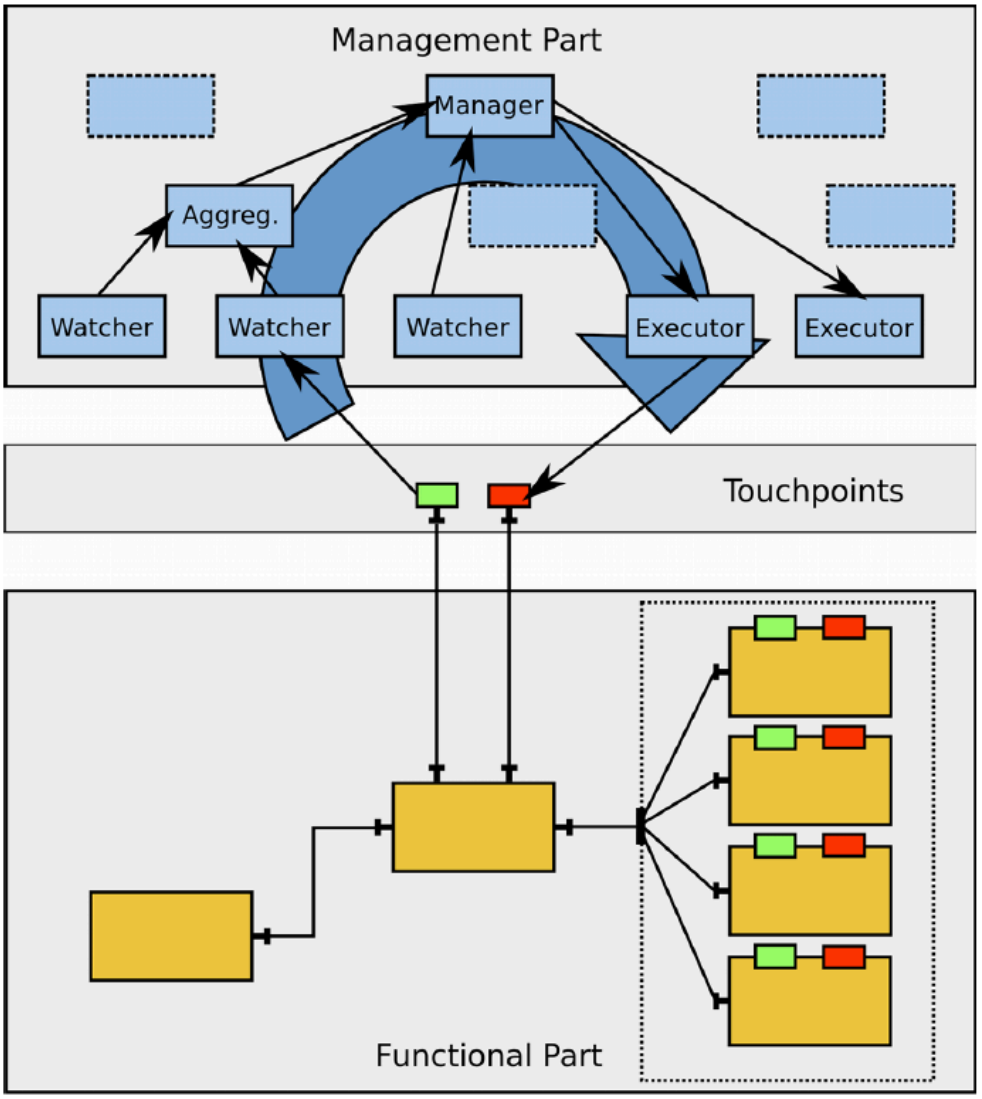


Рисунок 3.2 – Ієрархія елементів управління

Виконавці отримують команди від менеджерів і видають команди актуаторам, які діють на компоненти через інтерфейси керування. Датчики та виконавчі механізми взаємодіють із функціональними компонентами через інтерфейси керування (наприклад, контролери життєвого циклу та керування), тоді як керуючі елементи зазвичай обмінюються подіями за допомогою служби pub/sub, що надається середовищем виконання. Для керування та доступу до служб середовища виконання ME використовують

інтерфейс ActuatorInterface, прив’язаний до середовища виконання, який надає корисні послуги та методи контролю, такі як виявлення, розподіл, скасування розподілу, розгортання, пошук, прив’язування, скасування прив’язки, підписка та скасування підписки для публікації подій ME використовують інтерфейс TriggerInterface середовища виконання. Обидва клієнтські інтерфейси, ActuatorInterface і TriggerInterface, які використовуються ME, автоматично прив’язуються до відповідних серверних інтерфейсів середовища виконання під час розгортання (створення) ME. Щоб отримувати події, ME має реалізувати інтерфейс сервера EventHandlerInterface.

3.2 Програмування функціональних компонентів і груп компонентів розподіленої системи

При розробці самокерованої розподіленої компонентної програми з розробник виконує наступні кроки:

1. Розробка архітектури функціональної та управлінської частин програми. Цей крок включає наступну роботу: визначення та проектування функціональних компонентів (включаючи інтерфейси сервера та клієнта) та груп компонентів, призначення імен компонентам та інтерфейсам, визначення прив’язок компонентів та груп, визначення та проектування елементів керування, включаючи алгоритми обробників подій. для конкретних цілей управління програмою, визначення специфічних для програми моніторингу та подій активації, вибір попередньо визначених подій, виданих середовищем виконання, визначення джерел подій і підписок.

2. Опис (початкової) архітектури функціональних і управлінських частин в ADL, включаючи компоненти, їхні інтерфейси та прив’язки. Зауважте, що немає необхідності описувати всю конфігурацію в ADL, оскільки компоненти, групи та елементи керування можна також розгортати

та налаштовувати програмно за допомогою API активації, а не інтерпретатора ADL.

3. Програмування функціональних та управлінських компонентів. На цьому етапі розробник визначає класи та інтерфейси функціональних і управлінських компонентів, реалізує серверні інтерфейси (функціональні), обробники подій (управління), інтерфейси керування, наприклад, контролери життєвого циклу та прив’язки.

4. Програмування компонента (запуску), який завершує початкове розгортання та налаштування, виконане інтерпретатором ADL. Початкова частина програми (включаючи компонент запуску), описана в ADL на кроці 2, має бути розгорнута інтерпретатором ADL тоді як інша частина має бути розгорнута та налаштована компонентом запуску, визначеним програмістом, за допомогою інтерфейсу керування ActuatorInterface системи виконання.

Завершення розгортання може бути або тривіальним, якщо ADL максимально використовується на кроці 2, або складним, якщо досить невелика частина програми описана в ADL на кроці 2. Як правило, компонент запуску запрограмовано на виконання таких дій: bind компоненти, розгорнуті за допомогою ADL, виявлення та розподіл ресурсів (контейнерів) для розгортання компонентів; створювати, налаштовувати та прив'язувати компоненти та групи; створювати та налаштовувати елементи керування та підписувати їх на події; і початкові компоненти. Розглянемо, як наведені вище концепції застосовуються на практиці в програмуванні простого клієнт-серверного додатку HelloWorld (рис. 3.1), який є складеним компонентом, що містить два підкомпоненти, клієнт і сервер. Програма надає єдину службу, яка друкує повідомлення (привітання « Hello World!»), указане у виклику клієнта. У цьому прикладі серверний компонент надає серверний інтерфейс типу Service, що містить метод друку. Клієнтський компонент має клієнтський інтерфейс типу Service та серверний інтерфейс типу Main, що містить основний метод. Клієнтський інтерфейс клієнтського компонента прив’язаний до серверного інтерфейсу сервісного компонента. Композитний

компонент HelloWorld забезпечує інтерфейс сервера, який експортує відповідний інтерфейс клієнтського компонента; його головний метод викликається під час запуску програми. Примітивні компоненти реалізовані як класи Java, які реалізують серверні інтерфейси (наприклад, Service і Main у прикладі HelloWorld), а також будь-які необхідні інтерфейси керування (BindingController). Клас клієнтського компонента під назвою ClientImpl реалізує основний інтерфейс. Оскільки клієнтський компонент має клієнтський інтерфейс, який прив’язується до сервера, клас також реалізує інтерфейс BindingController, який є основним інтерфейсом керування для керування прив’язками. У наступному фрагменті коду представлено клас ClientImpl, який реалізує інтерфейси основного та контролера прив’язки. Зауважте, що сервісу клієнтського інтерфейсу присвоєно ім’я «s»:

```
public class ClientImpl implements Main, BindingController {
    // Client interface to be bound to server interface of Server component
    private Service service;
    private String citfName = "s"; // Name of the client interface
    // Implementation of the Main interface
    public void main (final String[] args) {
        // call the service to print the greeting
        service.print ("Hello world!");
     }
    // All methods below belong to the Binding Controller
    // interface with the default implementation
    // Returns names of all client interfaces of the component
    public String[] listFc ( ) {
        return new String[] { citfName };


     }
     // Returns the interface to which the given client interface is bound
     public Object lookupFc(final String citfName)
                                        throws NoSuchInterfaceException {
         if (!this.citfName.equals(citfName))
            throw new NoSuchInterfaceException(itfName);
         return service;
     }
     // Binds the client interface with the given name
     // to the given server interface
     public void bindFc(final String citfName, final Object sItf)
                                            throws NoSuchInterfaceException {
         if (!this.citfName.equals(citfName))
            throw new NoSuchInterfaceException(itfName);
         service = (Service)sItf;
     }
     // Unbinds the client interface with the given name
     public void unbindFc (final String citfName)
                                            throws NoSuchInterfaceException {
         if (!this.citfName.equals(citfName))
            throw new NoSuchInterfaceException(itfName);
         service = null;
     }
 }
```

Найпростіший спосіб зібрати компоненти — за допомогою ADL, який визначає набір компонентів, їх прив’язки та зв’язки між ними, і може бути використаний для автоматичного розгортання системи Fractal. Основними поняттями ADL є визначення компонентів, компоненти, інтерфейси та прив’язки. Опис ADL програми HelloWorld із однотонною службою такий:

```
<definition name="HelloWorld">
    <interface name="m" role="server" signature="Main"/>
    <component name="client">
      <interface name="m" role="server" signature="Main"/>
        <interface name="s" role="client" signature="Service"/>
        <content class="ClientImpl"/>
  </component>
    <component name="server">
        <interface name="s" role="server" signature="Service"/>
        <content class="ServerImpl"/>


     </component>
     <binding client="this.m" server="client.m" />
     <binding client="client.s" server="server.s" />
 </definition>
```

3.2.1 Групи компонентів і групові прив’язки

Прив’язки підтримують зв’язок між компонентами, розміщеними на різних машинах. Окрім зв’язків один до одного, які ми бачили раніше, також підтримуються групи та групові зв’язки, які особливо корисні для створення децентралізованих, стійких до збоїв програм. Групові прив’язки дозволяють розглядати набір компонентів, групу, як єдину сутність, і можуть доставляти виклики або всім членам групи (семантика «один до всіх»), або будь-якому випадково вибраному члену групи (один до- будь-яка семантика). Групи є динамічними, оскільки членство в них може змінюватися з часом (наприклад, збільшити розмір групи, щоб впоратися зі збільшеним навантаженням на рівні).

Керування групами здійснюється через API, який підтримує створення груп, прив’язування груп і компонентів, а також додавання/видалення членів групи. Крім того, Fractal ADL було розширено, щоб дозволити описувати групи як частину архітектури системи.

На рисунку малюнку 3.3 зображено додаток HelloGroup, у якому клієнтський компонент підключено до групи з двох компонентів служби без збереження стану (server1 і server2), використовуючи семантику виклику «один до будь-якого». Група компонентів служби надає службу, яка друкує повідомлення «Hello World!» привітання будь-ким із учасників групи на прохання клієнта.

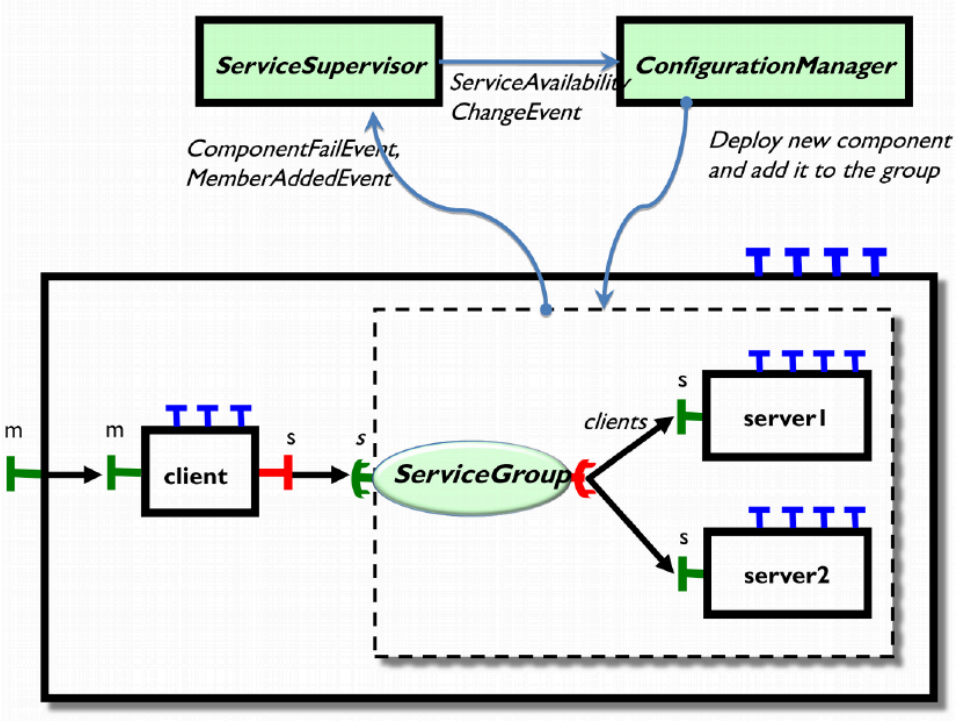


Рисунок 3.3 – Додаток в якому клієнтський компонент підключено до групи з двох компонентів служби

Початкова конфігурація цього прикладу програми (без елементів керування) може бути описана в ADL наступним чином:

```
<definition name="HelloGroup">
    <interface name="m" role="server" signature="Main"/>
    <component name="client">
        <interface name="m" role="server" signature="Main"/>
        <interface name="s" role="client" signature="Service"/>
        <content class="ClientImpl"/>
```

```
    </component>
    <component name="ServiceGroup">
        <interface name="s" role="server" signature="Service"/>
        <interface name="clients" role="client" signature="Service"
                                  cardinality="collection"/>
        <content class="GROUP"/>
  </component>
    <component name="server1">
    <interface name="s" role="server" signature="Service"/>
    <content class="ServerImpl"/>
  </component>
  <component name="server2">
    <interface name="s" role="server" signature="Service"/>
    <content class="ServerImpl"/>
  </component>
  <binding client="this.r" server="client.r" />
  <binding client="client.s" server="group.s" bindingType="groupAny"/>


    <binding client="group1.clients1" server="server1.s"/>
    <binding client="group1.clients2" server="server2.s"/>
 </definition>
```

Як видно з цього опису, група послуг представлена спеціальним компонентом із вмістом «GROUP». Тоді членство в групі представляється як прив'язка серверних інтерфейсів членів до клієнтських інтерфейсів групи. Атрибут bind-ingType представляє семантику і nvocation (у цьому випадку один до будь-якого). Групи також можна створювати та прив'язувати програмно за допомогою API активації (а саме клієнтського інтерфейсу ActuatorInterface, прив'язаного до часу виконання системи). Як приклад, наступний фрагмент коду Java ілюструє створення групи, виконане елементом керування:

```
// Code fragment from the StartManager class
// References to the Niche runtime interfaces
// bound on init or via binding controller
private NicheIdRegistry nicheIdRegistry;
private NicheActuatorInterface myActuatorInterface;
...


// Lookup the client component and all server components by names
ComponentId client =
               (ComponentId) nicheIdRegistry.lookup("HelloGroup _0/client");
ArrayList<ComponentId> servers = new ArrayList();
servers.add((ComponentId) nicheIdRegistry.lookup("HelloGroup _0/server1");
servers.add((ComponentId) nicheIdRegistry.lookup("HelloGroup_0/server2");
```

```
// Create a group containing all server components.
GroupId groupTemplate = myActuatorInterface.getGroupTemplate();
groupTemplate.addServerBinding("s", JadeBindInterface.ONE_TO_ANY);
GroupId serviceGroup = myActuatorInterface.createGroup(groupTemplate, servers);
// Bind the client to the group with one-to-any binding
myActuatorInterface.bind(client, "s", serviceGroup,
                         "s", JadeBindInterface.ONE_TO_ANY);
```

3.2.2 Програмування елементів управління

Керівна частина програмується за допомогою абстракцій елемента керування (ME), які включають датчики, спостерігачі, агрегатори, менеджери, виконавці та виконавчі механізми. ME зазвичай є реактивними компонентами, керованими подіями; тому розробка ME здебільшого полягає в програмуванні обробників подій, тобто методів серверного інтерфейсу EventHandlerInterface, які кожен ME повинен реалізувати, щоб отримувати події датчиків (включаючи визначені користувачем події та попередньо визначені події, видані системою виконання) та події від інших ME. Обробник події врешті-решт викликається, коли відповідна подія публікується (генерується). Обробники подій можуть бути запрограмовані на отримання та обробку подій різних типів. Типовий алгоритм керування обробником подій включає, але не обов’язково та не обмежується цим, послідовність умовних операторів керування if-then (else або else-if) (правила логіки керування), які перевіряють умови правила (пункт IF) на основі інформації, отриманої з отриманих подій, або/та її внутрішнього стану (що, у свою чергу, відображає попередні отримані події як частину моніторингової діяльності); приймати управлінське рішення та виконувати дії з керування та видавати події (речення THEN).

Під час програмування класу ME програміст повинен реалізувати такі три інтерфейси сервера: інтерфейс InitInterface для ініціалізації екземпляра ME, інтерфейс EventHandlerInterface для отримання та обробки подій; і інтерфейс MovableInterface для отримання контрольної точки, коли ME переміщується та повторно розгортається для реплікації або міграції

(контрольна точка передається новому екземпляру через його InitInterface). Щоб виконувати керуючі дії, підписуватися та публікувати події, клас ME повинен містити наступні два клієнтські інтерфейси:

- інтерфейс ActuatorInterface, який називається «актуатор»;
- інтерфейс TriggerInterface під назвою «тригер».

Обидва клієнтські інтерфейси прив’язані до системи середовища виконання, коли ME розгортається через його InitInterface або через інтерфейс BidingController.

Під час розробки коду керування ME (обробники подій) для керування функціональною частиною програми та підписки на події програміст використовує методи клієнтського інтерфейсу ActuatorInterface, який включає ряд методів активації, таких як виявлення, виділення, дезактивація. -виділити, розгорнути, створити а група компонентів, додати учасника до групи, прив’язати, роз’єднати, підписатися, скасувати підписку. Зауважте, що програміст може підписатися/скасувати підписку на попередньо визначені вбудовані події (наприклад, збій компонента, зміна членства в групі), видані вбудованими датчиками системи виконання. Для публікації подій програміст використовує клієнтський інтерфейс TriggerInterface ME.

Наприклад, на рис. 3.4 зображено програму HelloGroup, яка надає групову службу з можливостями самовідновлення. Контроль зворотного зв’язку в додатку підтримує розмір групи (зазначену мінімальну кількість компонентів служби) незважаючи на збої вузла, тобто якщо будь-який із компонентів у групі виходить з ладу, створюється новий компонент служби та додається до групи, щоб група завжди містить задану кількість серверів. Контур керування самовідновленням включає агрегатор Supervisor, який відстежує кількість компонентів у групі, і менеджера конфігурації, який відповідає за створення та додавання нового компонента служби за запитом від керівника служби.

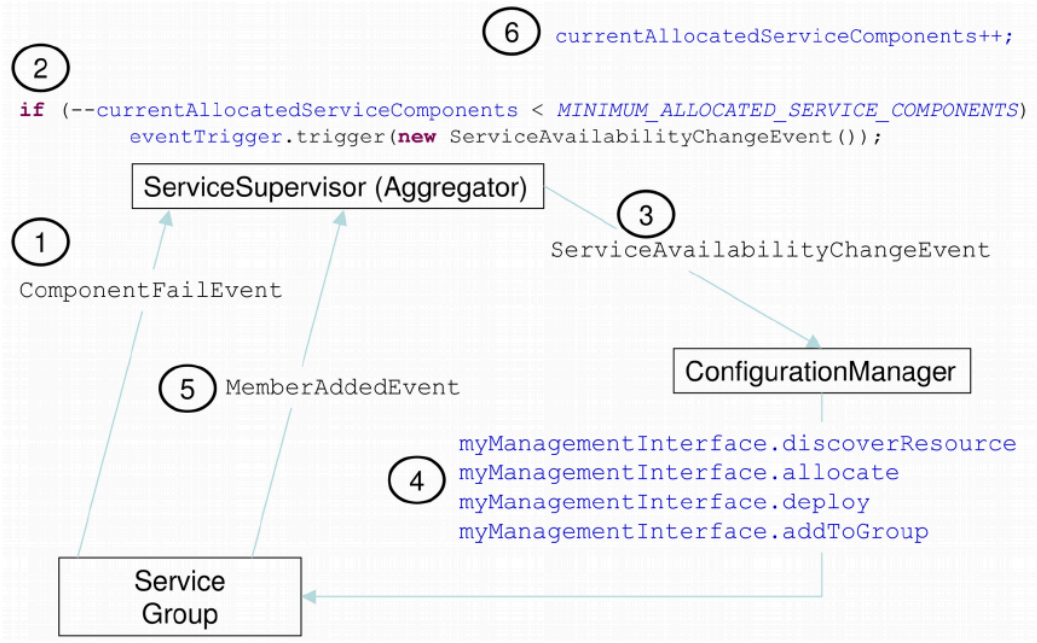


Рисунок 3.4 – Події та дії в циклі самовідновлення програми

На рисунку 3.4 зображено послідовність подій і керуючих дій компонентів управління. Зокрема, якщо один із сервісних компонентів групи сервісів виходить з ладу, груповий датчик видає подію про відмову компонента, отриману сервіс-супервайзером (1), який перевіряє, чи кількість компонентів впала нижче заданого порогового значення (2). Якщо так, наглядач сервера запускає подію Service-Availability-Change, отриману диспетчером конфігурації (3), яка виліковує компонент, тобто створює новий екземпляр серверного компонента та додає його до групи (4). Коли до групи додається новий учасник, сервіс-супервізор, який відстежує кількість компонентів сервера, отримує сповіщення за допомогою попередньо визначеної події Member-Added-Event, виданої датчиком групи (5, 6). Наведений нижче скорочений фрагмент коду Java показує логіку керування диспетчера конфігурацій, відповідального за відновлення несправного компонента сервера після отримання події Service-Availability-Change, створеної Supervisor Service (кроки 3 і 4 на рисунку 3.4):

```
        nodeRequirements = DISCOVER_PREFIX + initAttributes[2];
}
...
 // event handler, invoked on an event
public void eventHandler(Serializable e, int flag) {
        // For any case, check event type,
        // ignore if it is not the event of interest (should not happen)
        if (! (e instanceof ServiceAvailabilityChangeEvent)) return;
        // Find a node that meets the requirements for a server component.

      try {
          newNode =
             myManagementInterface.oneShotDiscoverResource( nodeRequirements);
      } catch (OperationTimedOutException err) {
          ... // Retry later (the code is removed)
      }
      // Allocate resources for a server component at the found node.
      try {
          List allocatedResources =
                                  myManagementInterface.allocate(newNode, null);
      } catch (OperationTimedOutException err) {
          ... // Retry later (the code is removed)
      }
       ...
      String deploymentParams = Serialization.serialize(serviceCompProps);
      // Deploy a new server component instance at the allocated node.


   try {
       deployedComponents = myManagementInterface.deploy( allocatedResource,
                                                          deploymentParams );
   } catch (OperationTimedOutException err) {
   ... // Retry later (the code is removed)
   }
   ComponentId cid = (ComponentId)((Object[])deployedComponents.get(0))[1];
   // Add the new server component to the service group and start the server
   myManagementInterface.update(componentGroup, cid,
                NicheComponentSupportInterface.ADD_TO_GROUP_AND_START);
   }
```

У той час як ME взаємодіють один з одним здебільшого за допомогою подій, датчики та виконавчі механізми запрограмовані на взаємодію з функціональними компонентами через прив’язки інтерфейсу. Інтерфейси між датчиками та компонентами визначаються програмістом, який може використовувати методи взаємодії між датчиком і компонентом або натисканням, або витягуванням. У випадку push-методу компонент надсилає датчику надсилати подію. У цьому випадку інтерфейс клієнта компонента прив’язаний до інтерфейсу сервера відповідного датчика. У випадку методу

витягування датчик витягує стан із компонента. У цьому випадку клієнтський інтерфейс датчика прив’язаний до серверного інтерфейсу відповідного компонента. Датчик і компонент автоматично пов’язуються, коли датчик розгортається спостерігачем. Приведення в дію (керуючі дії) може здійснюватися ME або через приводи, прив’язані до функціональних компонентів, або безпосередньо на компонентах через їхні інтерфейси керування за допомогою API активації. Актуатори програмуються подібно до датчиків і розгортаються виконавцями. За аналогією з датчиками, виконавчий механізм можна запрограмувати на взаємодію з контрольованим компонентом у спосіб штовхання та/або витягування. У першому випадку (штовхання) привід проштовхує компонент через інтерфейси керування компонентом, які можуть бути або інтерфейсами для конкретної програми, визначеними програмістом, або інтерфейсами керування Fractal, наприклад, LifeCycleController і AttributeController. У разі приведення в дію на основі витягування, керований компонент перевіряє свій привод на наявність дій, які необхідно виконати.

3.2.3 Розгортання та управління ресурсами

Пропонована система підтримує розгортання компонентів і керування ресурсами за допомогою концепцій пакета компонентів і вузла. Пакет компонентів — це пакет, який містить виконувані файли, необхідні для створення компонентів, дані, необхідні для їх правильного функціонування, а також метадані, що описують їхні властивості. Вузол — це фізична або віртуальна машина, на якій розгортаються та виконуються компоненти. Вузол забезпечує обробку, зберігання та комунікаційні ресурси, які спільно використовуються між розгорнутими компонентами.

Система надає основні примітиви для виявлення вузлів, розподілу ресурсів на цих вузлах і розгортання компонентів; ці примітиви розроблені, щоб сформувати основу для зовнішніх служб для розгортання компонентів і

керування їхніми основними ресурсами. У поточному прототипі пакети компонентів є пакетами OSGi і керовані ресурси включають процесорний час, фізичну пам'ять, простір для зберігання та пропускну здатність мережі. Fractal ADL було розширено, щоб дозволити вказувати пакети та обмеження ресурсів на вузлах. Ці розширення проілюстровано в наведеному нижче екстракті ADL, який уточнює опис клієнта та складений опис у прикладі HelloGroup (додані елементи виділено жирним шрифтом):

```
<definition name="HelloGroup">
    <interface name="m" role="server" signature="Main"/>
    <component name="client">
    <interface name="m" role="server" signature="Main"/>
    <interface name="s" role="client" signature="Service"/>
    <content class="ClientImpl"/>
    <packages>
              <package name="ClientPackage v1.3" >
                    <property name="local.dir" value="/tmp/j2ee"/>
            </package>
    </packages>
    <virtual-node name="node1" resourceReqs="(&(memory>=1)(CPUSpeed>=1))"/>
    </component>
 <!-- description of other components and bindings  (is not shown) -->
  ...
    <virtual-node name="node1">
</definition>
```

Елемент packages надає інформацію про пакети OSGi, необхідні для створення компонента. Пакети ідентифікуються своїм унікальним іменем у сховищі пакетів OSGi (наприклад, «ClientPackage v1.3»). Елемент віртуального вузла описує вимоги до ресурсів і розташування компонентів. Під час розгортання кожен віртуальний вузол зіставляється з вузлом (контейнером), який відповідає заданим вимогам до ресурсів, указаним в атрибуті resourceReqs. Потім на цьому вузлі встановлюються необхідні пакети та створюється відповідний компонент. У цьому прикладі клієнтський і складений компоненти розташовані разом на вузлі з пам’яттю більше 1 ГБ і частотою ЦП більше 1 ГГц.

3.2.4 Ініціалізація коду управління

ADL включає підтримку ініціалізації частини керування програмою у формі компонентів менеджера запуску. Початкові менеджери мають попередньо визначене визначення «StartManagementType», яке містить набір клієнтських інтерфейсів, відповідних API. Ці інтерфейси неявно прив'язуються системою після створення екземплярів менеджерів запуску. Оголошення менеджера запуску демонструється в наведеному нижче екстракті ADL, який уточнює приклад HelloGroup:

```
<component name="StartManager" definition="org.ow2.jade.StartManagementType">
   <content class=" helloworld.managers.StartManager"/>
 </component>
```

Як правило, менеджер запуску містить код для створення, налаштування та активації набору елементів керування, які складають частину керування додатком. У прикладі HelloGroup частина керування реалізує поведінку самовідновлення та покладається на агрегатор і менеджер, який контролює групу серверів і підтримує її розмір, незважаючи на збої вузлів. Потім реалізація менеджера запуску (клас StartManager) містить код для розгортання та налаштування елементів циклу самовідновлення, показаного на рисунку 3.3 (тобто ServiceSupervisor і ConfigurationManager). Код фактично знаходиться в реалізації інтерфейсу LifeCycleController (операція startFc) менеджера запуску, як показано далі:

```
// Code fragment from the StartManager class of the HelloGroup application
public class StartManager implements BindingController, LifeCycleController {
// References to the Niche runtime interfaces
// bound on init or via binding controller
private NicheIdRegistry nicheIdRegistry;
private NicheActuatorInterface myActuatorInterface;
...
// Invoked by the Niche runtime system
public void startFc() throws IllegalLifeCycleException {
        ...
    // Lookup client and servers, create service group
    // and bind client to the group (code is not shown)
    GroupId serviceGroup = myActuatorInterface.createGroup(...);
 ...
    // Configure and deploy the Service Supervisor aggregator
  GroupId gid = serviceGroup;
  ManagementDeployParameters params = new ManagementDeployParameters();
    params.describeAggregator( ServiceSupervisor.class.getName(), "SA", null,
                                        new Serializable[] { gid.getId() } );
```

```
    NicheId serviceSupervisor =
                      myActuatorInterface.deployManagementElement(params, gid);
    // Subscribe the aggregator to events from group
    myActuatorInterface.subscribe(gid, serviceSupervisor,
                                  ComponentFailEvent.class.getName());
    myActuatorInterface.subscribe(gid, serviceSupervisor,
                                  MemberAddedEvent.class.getName());
    // Configure and deploy the Configuration manager
 String minimumNodeCapacity = "200";
 params = new ManagementDeployParameters();
    params.describeManager(ConfigurationManager.class.getName(), "CM", null,
                           new Serializable[] { gid, fp, minimumNodeCapacity });
    NicheId configurationManager =
                     myActuatorInterface.deployManagementElement( params, gid );
    // Subscribe the manager to events from the aggregator
    myActuatorInterface.subscribe(serviceSupervisor, configurationManager,
                                  ServiceAvailabilityChangeEvent.class.getName());
  ...
}
```

3.3 Методологія проектування для самоуправління в розподілених обчислювальних середовищах

Самокеровану програму можна розкласти на три частини: функціональну частину, точки дотику та частину керування. Процес проектування починається з визначення функціональних вимог і вимог до управлінської частини відповідно. У випадку ніші функціональна частина програми розроблена шляхом визначення інтерфейсів, компонентів, груп компонентів і прив’язок. Частина керування розроблена на основі вимог до управління, шляхом визначення автономних менеджерів (елементів керування) та необхідних точок дотику (датчиків і виконавчих механізмів). Touchpoints дозволяють керувати функціональною частиною, тобто роблять її керованою.

Автономний менеджер — це цикл керування, який постійно контролює та впливає на функціональну частину програми, коли це необхідно. Для багатьох програм у середовищі бажано розкласти автономний менеджер на кілька взаємодіючих автономних менеджерів, кожен з яких виконує певну функцію управління та/або контролює певну частину програми. Декомпозиція управління може бути мотивована різними причинами, такими

як: Це дозволяє уникнути єдиної точки відмови. Може знадобитися розподіл накладних витрат на керування між залученими ресурсами. Для самостійного керування складною системою може знадобитися більше ніж один автономний менеджер для спрощення дизайну шляхом поділу проблем. Декомпозицію також можна використовувати для підвищення продуктивності управління шляхом одночасного виконання різних завдань управління та розміщення автономних менеджерів ближче до ресурсів, якими вони керують.

Ми визначаємо наступні ітераційні кроки, які необхідно виконати під час проектування та розробки частини керування самокерованою розподіленою програмою децентралізованим способом, враховуючи вимоги до керування та точки дотику.

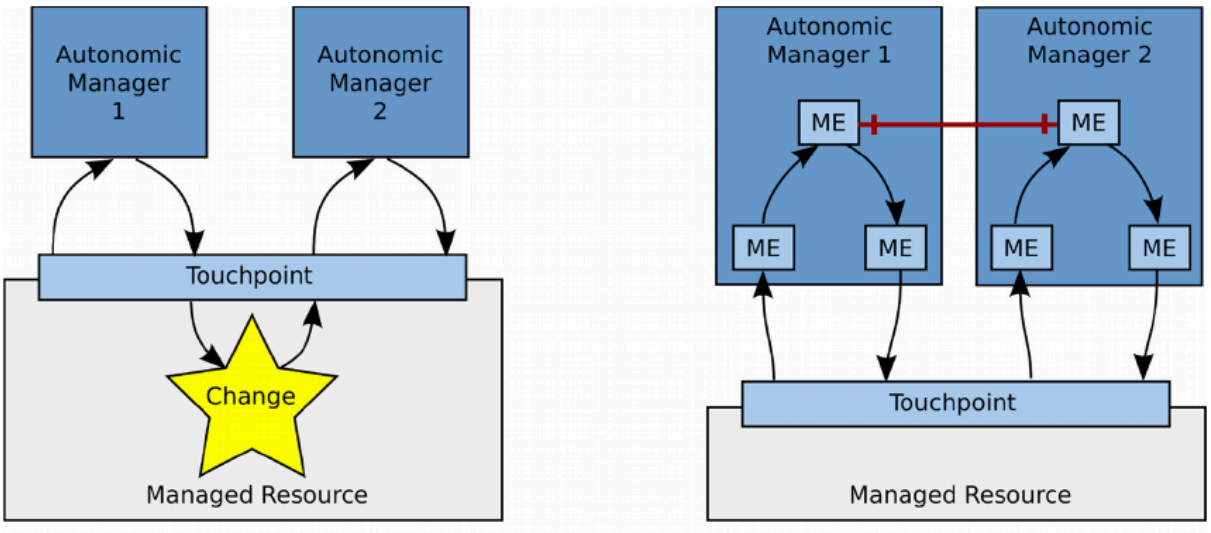
- Декомпозиція: Першим кроком є розділення логіки управління на кілька завдань управління. Декомпозиція може бути або функціональною (наприклад, завдання визначаються на основі того, які самовластивості вони реалізують), або просторовою (наприклад, завдання визначаються на основі структури керованої програми). Основна проблема дизайну, яку слід розглянути на цьому кроці, — це деталізація завдань, припускаючи, що завдання або групу пов’язаних завдань може виконувати один керівник.
- Призначення: завдання потім призначаються автономним менеджерам, кожен з яких стає відповідальним за одне або більше завдань управління. Призначення можна виконати на основі властивостей self, до яких належить завдання (згідно з функціональною декомпозицією), або на основі того, до якої частини програми це завдання відноситься (згідно з просторовою декомпозицією).
- Оркестровка: хоча автономні менеджери можуть бути розроблені незалежно, кілька автономних менеджерів, у загальному випадку, не є незалежними, оскільки вони керують тією самою системою та існують залежності між завданнями управління. Тому їм необхідно взаємодіяти та координувати свої дії, щоб уникнути конфліктів і втручань і належним чином

керувати системою. Оркестровка автономних менеджерів обговорюється в наступному розділі.

- Відображення: набір автономних менеджерів потім відображаються на ресурси, тобто на вузли розподіленого середовища. Основним питанням, яке слід розглянути на цьому кроці, є оптимізоване розміщення менеджерів і, можливо, функціональних компонентів на вузлах, щоб покращити ефективність управління.

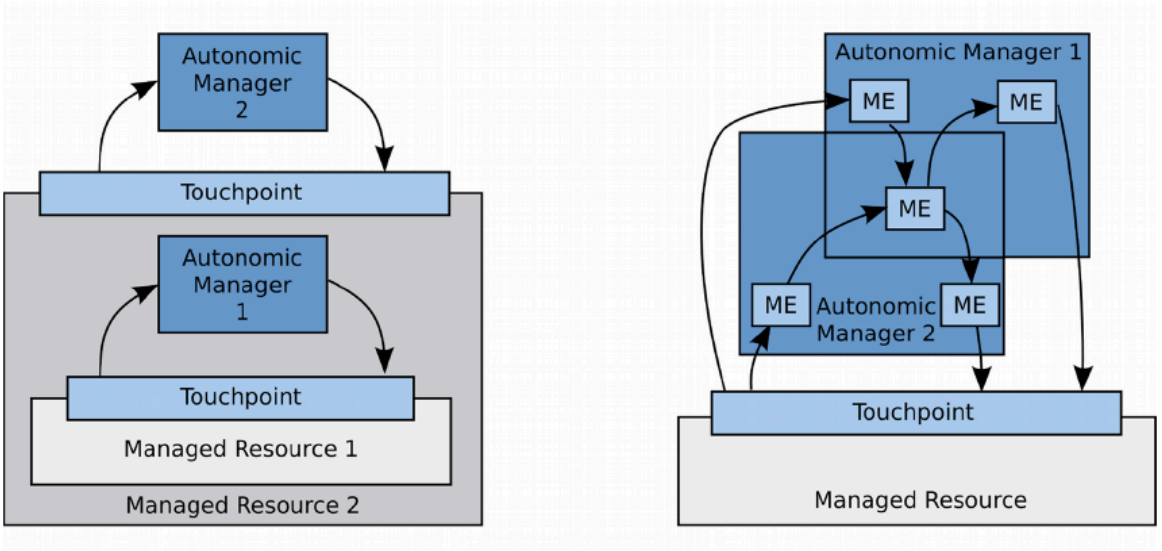
У цьому підрозділі наша основна увага приділяється оркестровці автономних менеджерів як найскладнішій і менш вивченій проблемі. Дії та цілі інших етапів більше пов’язані з класичними проблемами розподілених систем, такими як розділення та відокремлення завдань, а також оптимальне розміщення модулів у розподіленому середовищі.

Автономні менеджери можуть взаємодіяти та координувати свою роботу наступними чотирма способами, як обговорено нижче та показано на рисунках 3.5 і 3.6: непрямі взаємодії через керовану систему (стигмергія); ієрархічна взаємодія (через точки дотику); пряма взаємодія (через прямі зв'язки); спільне використання елементів управління.



а) ефект стигметрії б) пряма взаємодія

Рисунок 3.5 – Шаблони взаємодії



а) ієрархічне управління б) спільні елементи керування

Рисунок 3.6 – Шаблони взаємодії

Стигмергія. Стигмергія – це спосіб непрямого спілкування та координації між агентами. Агенти вносять зміни у своє середовище, і ці зміни відчуваються іншими агентами та змушують їх виконувати більше дій. У нашому випадку агенти є автономними менеджерами, а середовище є керованим додатком.

Ефект стигмергії, як правило, неминучий, якщо у вас є більше ніж один автономний менеджер і може спричинити небажану поведінку під час виконання. Прихована стигмергія ускладнює розробку самокерованої системи з декількома автономними менеджерами. Однак стигмергія може бути частиною дизайну та використовуватися як спосіб оркестрування автономних менеджерів.

Ієрархічне управління. Під ієрархічним управлінням ми маємо на увазі, що деякі автономні менеджери можуть контролювати та контролювати інших автономних менеджерів. Автономні менеджери нижчого рівня вважаються керованим ресурсом для автономного менеджера вищого рівня. Комунікації між рівнями відбуваються за допомогою точок дотику.

Менеджери вищого рівня можуть відчувати та впливати на керівників нижчого рівня.

Автономні менеджери на різних рівнях часто працюють у різних часових масштабах. Автономні менеджери нижчого рівня використовуються для управління змінами в системі, які потребують негайних дій. Автономні менеджери вищого рівня часто повільніші та використовуються для регулювання та оркестрування системи шляхом моніторингу глобальних властивостей і відповідного налаштування автономних менеджерів нижчого рівня.

Пряма взаємодія. Автономні менеджери можуть безпосередньо взаємодіяти один з одним. Технічно це досягається прямим зв'язком (через прив'язки або події) між відповідними елементами управління в автономних менеджерах. Перехресні прив'язки автономних менеджерів можна використовувати для координації автономних менеджерів і уникнення небажаної поведінки, наприклад умов перегонів або коливань.

Спільні елементи керування. Інший спосіб для автономних менеджерів спілкуватися та координувати свої дії – це спільне використання елементів управління. Це можна використовувати для обміну станом (знанням) і для синхронізації їхніх дій.

3.4 Імплементація концепції та моделей самокерованих додатків

Щоб продемонструвати нашу методологію проектування, ми представляємо дві самокеровані служби:

- надійна служба зберігання під назвою YASS (Yet Another Storage Service);
- надійний обчислювальний сервіс під назвою YACS (Yet Another Computing Service).

Кожна зі служб має можливості самовідновлення та самоконфігурації та може виконуватися в динамічному розподіленому середовищі, тобто

служби можуть працювати, навіть якщо комп’ютери приєднуються, залишають або виходять з ладу в будь-який час. Кожен із сервісів реалізує відносно прості алгоритми самоконтролю, які можна розширити до більш складних, повторно використовуючи існуючий код моніторингу та активації сервісів.

YASS (Yet Another Storage Service) — це надійна служба зберігання, яка дозволяє клієнту зберігати, читати та видаляти файли на кількох комп’ютерах. Сервіс прозоро копіює файли, щоб досягти високої доступності файлів і скоротити час доступу. YASS підтримує вказану кількість реплік файлів, незважаючи на те, що вузли залишають або виходять з ладу, і вона може масштабуватися (тобто збільшувати доступний простір для зберігання), коли загальна вільна пам’ять є нижчою за вказане порогове значення. Завдання управління включають підтримку ступеня реплікації файлів; підтримання загального простору для зберігання та загального вільного простору; підвищення доступності популярних файлів; звільнення додаткового виділеного сховища; і балансування збережених файлів серед доступних ресурсів.

YACS (Yet Another Computing Service) — це надійна розподілена обчислювальна служба, яка дозволяє клієнту надсилати та виконувати завдання, які є пакетами незалежних завдань, у мережі вузлів (комп’ютерів). YACS гарантує виконання завдань,незважаючи на вихід або збій вузлів. YACS масштабується, тобто змінює кількість компонентів виконання, коли змінюється кількість робіт/завдань. YACS підтримує контрольні точки, що дозволяє перезапустити виконання з останньої контрольної точки, коли робочий компонент виходить з ладу або залишає роботу.

У цьому прикладі показано, як розробити самокеровану розподілену систему, яка контролюється та контролюється декількома розподіленими автономними менеджерами.

Припускаючи, що YASS має бути розгорнуто та надано в динамічному розподіленому середовищі, необхідні такі функції керування, щоб зробити

службу зберігання самокерованою за наявності динамічності ресурсів і навантаження: служба повинна терпіти відтік ресурсів (приєднання /leaves/failures), оптимізувати використання ресурсів і вирішити гарячі точки. Ми визначаємо наступні завдання на основі функціональної декомпозиції управління відповідно до властивостей self. а саме самовідновлення, самоконфігурація та само оптимізація.

Екземпляр YASS складається з зовнішніх компонентів і компонентів зберігання, як показано на рисунку 3.7. Інтерфейсний компонент забезпечує інтерфейс користувача, який використовується для взаємодії зі службою зберігання. Компоненти сховища представляють ємність сховища, доступну на ресурсі, на якому вони розгорнуті.

Компоненти зберігання об’єднані в групу зберігання. Користувач видає команди (зберігати, читати та видаляти), використовуючи інтерфейс. Запит на зберігання надсилається до довільного компонента сховища (використовуючи прив’язку «один до будь-якого» між інтерфейсом і групою сховища), який, у свою чергу, знаходить кілька r різних компонентів сховища, де r — це ступінь реплікації файлу, з достатньою кількістю вільного місця. місце для зберігання репліки файлу.

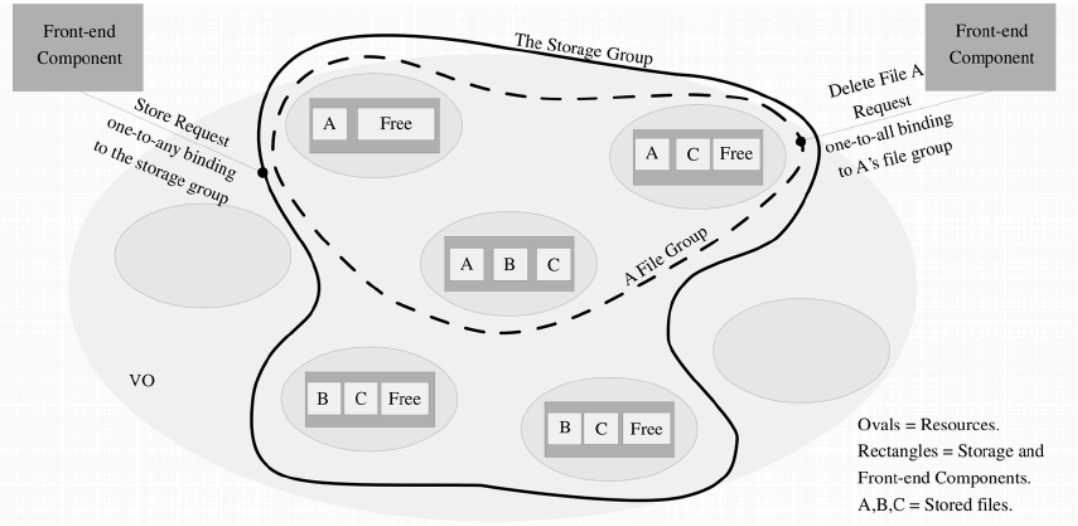


Рисунок 3.7 – Функціональний дизайн YASS

Враховуючи те, що функціональна частина YASS була розроблена, щоб керувати нею, нам потрібно забезпечити точки дотику. Система забезпечує базові точки дотику для маніпулювання архітектурою та ресурсами системи, такі як датчики збоїв ресурсів і створення груп компонентів; і приводи для розгортання та зв'язування компонентів. Для керування YASS у динамічному середовищі потрібні наступні автономні менеджери. Усі чотири техніки оркестровки, описані в попередньому розділі про методологію проектування, демонструються нижче.

Автономний диспетчер репліки: автономний диспетчер репліки відповідає за підтримку бажаного рівня реплікації для кожного збереженого файлу, незважаючи на збій ресурсів і їхнє зникнення. Цей автономний менеджер додає YASS властивість самовідновлення. Автономний менеджер репліки складається з двох елементів керування, File-Replica-Aggregator і File-Replica-Manager, як показано на рисунку 3.8. File-Replica-Aggregator відстежує групу файлів, що містить підмножину компонентів зберігання, які містять репліки файлів, підписуючись на події збою ресурсу або виходу, спричинені будь-яким із членів групи. Ці події надходять, коли ресурс, на якому розгорнуто член-компонент у групі, збирається залишити або стався збій. File-Replica-Aggregator реагує на ці події, запускаючи подію зміни репліки для File-Replica-Manager, який видає команду пошуку та відновлення репліки.

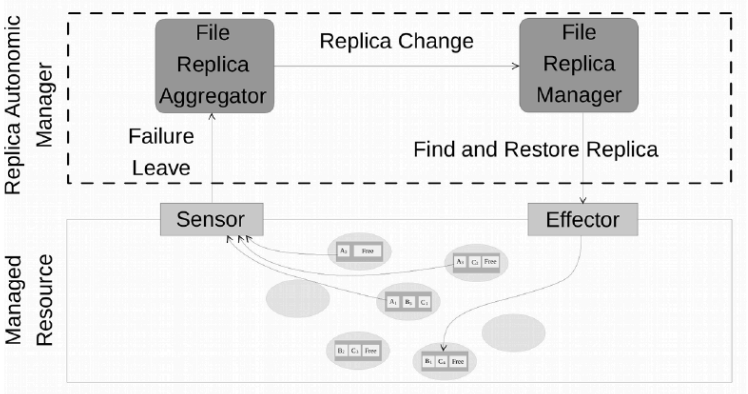


Рисунок 3.8 – Цикл керування самовідновленням для відновлення реплік файлів

Автономний диспетчер сховища: автономний диспетчер сховища відповідає за підтримку загальної ємності сховища та загального вільного простору в групі сховища за наявності динамічності, щоб відповідати вимогам QoS. Динамічність пов’язана з відмовою/залишенням ресурсів (впливає як на загальний, так і на вільний простір для зберігання), або через створення/додавання/видалення файлів (впливає лише на вільний простір для зберігання).

Автономний диспетчер сховища змінює конфігурацію YASS, щоб відновити загальний вільний простір або загальну ємність сховища відповідно до вимог. Реконфігурація виконується шляхом виділення вільних ресурсів і розгортання на них додаткових компонентів зберігання. Цей автономний менеджер додає властивість самоналаштування до YASS. Автономний менеджер сховища складається з Component-Load-Watcher, Storage-Ggregator і Storage-Manager, як показано на рисунку 3.9.

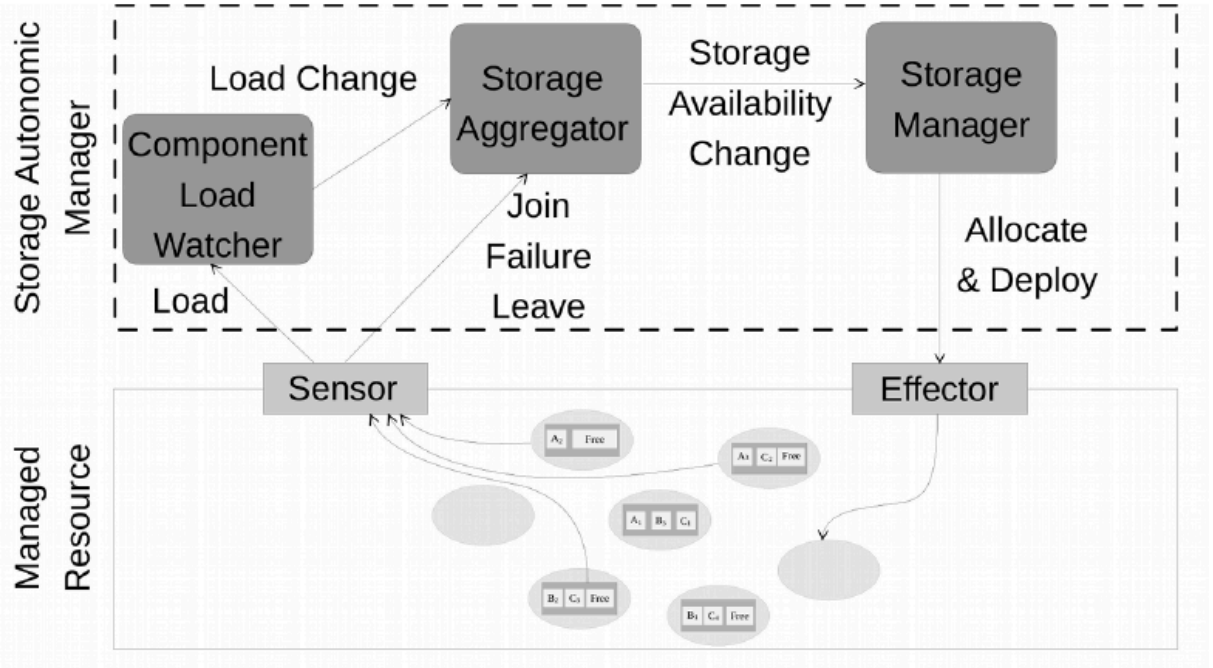


Рисунок 3.9 – Контур керування самоналаштуванням для додавання сховища

Component-Load-Watcher відстежує групу зберігання, що містить усі компоненти зберігання, на предмет змін загального доступного вільного простору шляхом підписки на події датчиків навантаження. Component-Load-Watcher ініціює подію зміни навантаження, коли навантаження змінюється на попередньо визначену дельту. Storage-Aggregator підписаний на подію зміни завантаження Component-Load-Watcher і події відмови ресурсу, виходу та приєднання (зверніть увагу, що File-Replica-Aggregator також підписується на події відмови ресурсу та виходу). Storage-Aggregator, аналізуючи ці події, зможе оцінити загальну ємність пам’яті та загальний вільний простір. Storage-Aggregator ініціює подію зміни доступності сховища, коли загальний і/або вільний простір сховища падає нижче попередньо визначеного порогу. Storage-Manager реагує на цю подію, намагаючись виділити більше ресурсів і розгорнути на них компоненти зберігання. Пряма взаємодія для координації автономних менеджерів: описані вище два автономні менеджери, репліка автономного менеджера та автономний менеджер зберігання, здаються незалежними. Перший менеджер відновлює файли, а інший менеджер відновлює сховище. Але можлива ситуація змагання між двома автономними менеджерами, яка призведе до збою репліки автономного менеджера.

Якби автономний диспетчер реплік чекав, поки автономний диспетчер сховища завершить роботу, він міг би відтворити репліки. Ми використали пряму взаємодію для координації двох автономних менеджерів, прив’язавши File-Replica-Manager до Storage-Manager.

Перед відновленням файлів File-Replica-Manager інформує Storage-Manager про обсяг пам’яті, необхідний для відновлення файлів. Диспетчер сховища перевіряє доступне сховище та інформує диспетчер файлів-реплік, що він може продовжити, якщо доступно достатньо місця, або попросити його зачекати. Використана тут пряма координація не означає, що один керівник контролює іншого. Наприклад, якщо від файлу залишилася лише одна репліка, File-Replica-Manager може проігнорувати запит

очікування від Storage-Manager і все одно продовжити відновлення файлу.

Висновки до розділу 3

Отже, в цьому розділі виконана побудова моделі та шаблонів взаємодії розподіленого обчислювального середовища в якому вирішено наступні завдання та проблеми:

- першим завданням є ефективне та надійне виявлення ресурсів. Усі ресурси (контейнери) можна виявити за допомогою накладання.
- друга проблема полягає в розробці надійної та ефективної інфраструктури зондування та активації. Для ефективності ми використовуємо механізм push, а не механізм витягування.
- третя проблема полягає в тому, щоб уникнути вузького місця в управлінні або єдиної точки відмови. Ми виступаємо за децентралізований підхід до управління.
- четверта проблема полягає в масштабі, під яким ми мали на увазі, що в динамічних системах швидкість змін (приєднання, вихід, збій ресурсів, зміна навантаження компонентів тощо) є високою, і що важливо зменшити потребу в спілкуванні в системі.

ВИСНОВКИ

В кваліфікаційній роботі виконано дослідження та імплементацію моделей покращення керованості великих масштабованих розподілених обчислювальних систем. Дослідження в даній роботі спрямовані на забезпечення та досягнення самоуправління для великомасштабних розподілених систем. У цьому дослідженні розглядаються проблеми забезпечення самоуправління для великомасштабних та динамічних розподілених систем, щоб приховати складність системи та автоматизувати її керування, організацію, налаштування та захист. Дана мета досягається впроваджуючи архітектуру автономних обчислень у повністю децентралізований спосіб, щоб відповідати вимогам великомасштабних розподілених систем. Автономна обчислювальна архітектура складається в основному з точок дотику (давачів) і автономних менеджерів, які спілкуються з керованою системою (через точки дотику) і один з одним для досягнення цілей управління. Представлено шаблони взаємодії кількох автономних менеджерів і пропонуємо кроки для проектування самоуправління у великих розподілених системах. Ми продовжуємо наші дослідження, вирішуючи проблеми надійності управління та узгодженості даних, яких не уникнути в розподіленій системі. Ми розробили децентралізований алгоритм, який гарантує надійність автономних менеджерів, дозволяючи їм витримувати безперервний відтік. Пропонований підхід базується на реплікації автономного менеджера за допомогою реплікації кінцевого автомата з повторно конфігурованим набором реплік. Розроблений алгоритм автоматизує реконфігурацію (міграцію) репліки встановлено таким чином, щоб витримувати безперервний відтік. Для узгодженості даних запропоновано дизайн і алгоритми для надійного відмовостійкого сховища ключів і значень на основі більшості, що підтримує кілька рівнів узгодженості, що базується на одноранговій мережі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, “A survey and com-parison of peer-to-peer overlay network schemes,” Communications Surveys & Tutorials, IEEE, vol. 7, pp. 72-93, Second Quarter 2005.

2. Foster, C. Kesselman, and S. Tuecke, “The anatomy of the grid: Enabling scalable virtual organizations,” Int. J. High Perform. Comput. Appl., vol. 15, pp. 200-222, Aug. 2001.

3. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “A view of cloud computing,” Commun. ACM, vol. 53, pp. 50-58, Apr. 2010.

4. J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” Computer, vol. 36, pp. 41-50, Jan. 2003.

5. P. Horn, “Autonomic computing: IBM’s perspective on the state of information technology,” Oct. 15 2001.

6. IBM, “An architectural blueprint for autonomic computing, 4th edition.” http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_ Blueprint_White_Paper_4th.pdf, June 2006.

7. E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani, “The fractal component model and its support in java: Experiences with autoadaptive and reconfigurable systems,” Softw. Pract. Exper., vol. 36, no. 11-12, pp. 1257-1284, 2006.

8. C. Arad, J. Dowling, and S. Haridi, “Building and evaluating P2P systems using the Kompics component framework,” in Peer-to-Peer Computing (P2P09), pp. 93-94, IEEE, Sept. 2009.

9. R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, “Serving large-scale batch computed data with project voldemort,” in The 10th USENIX Conference on File and Storage Technologies (FAST’12), February 2012.

10. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” SIGOPS Oper. Syst. Rev., vol. 44, pp. 35-40, Apr. 2010.

11. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: amazon’s highly available key-value store,” in Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, SOSP ’07, (New York, NY, USA), pp. 205-220, ACM, 2007.

12. P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, “Characterizing, modeling, and generating workload spikes for stateful services,” in Proceedings of the 1st ACM symposium on Cloud computing, SoCC ’10, (New York, NY, USA), pp. 241-252, ACM, 2010.

13. B. Trushkowsky, P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, “The scads director: scaling a distributed storage system under stringent performance requirements,” in Proceedings of the 9th USENIX conference on File and stroage technologies, FAST’11, (Berkeley, CA, USA), pp. 12-12, USENIX Association, 2011.

14. B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, “Pnuts: Yahoo!’s hosted data serving platform,” Proc. VLDB Endow., vol. 1, pp. 1277-1288, August 2008.

15. F. Dabek, A Distributed Hash Table. PhD thesis, Massachusetts Institute of Technology, November 2005.

16. P. V. Roy, S. Haridi, A. Reinefeld, J.-B. Stefani, R. Yap, and T. Coupaye, “Self management for large-scale distributed systems: An overview of the selfman project,” in FMCO ’07: Software Technologies Concertation on Formal Methods for Components and Objects, (Amsterdam, The Netherlands), Oct 2007.

17. S. White, J. Hanson, I. Whalley, D. Chess, and J. Kephart, “An architectural approach to autonomic computing,” in Autonomic Computing, 2004. Proceedings. International Conference on, ICAC2004, pp. 2-9, may 2004.

18. P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, “Composing adaptive software,” Computer, vol. 37, pp. 56-64, July 2004.

19. M. Parashar, Z. Li, H. Liu, V. Matossian, and C. Schmidt, Self-star Properties in Complex Information Systems, vol. 3460/2005 of Lecture Notes in Computer Science, ch. Enabling Autonomic Grid Applications: Requirements, Models and Infrastructure, pp. 273-290. Springer Berlin / Heidelberg, May 2005.

20. J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, Feedback Control of Computing Systems. John Wiley & Sons, September 2004.

21. Y. Diao, J. L. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung, “Self-managing systems: a control theory foundation,” in Proc. 12th IEEE International Conference and Workshops on the Engineering of Computer- Based Systems ECBS ’05, pp. 441-448, Apr. 4-7, 2005.

22. S. Abdelwahed, N. Kandasamy, and S. Neema, “Online control for self-management in computing systems,” in Proc. 10th IEEE Real-Time and Embedded Technology and Applications Symposium RTAS 2004, pp. 368-375, May 25-28, 2004.

23. R. J. Anthony, “Emergence: a paradigm for robust and scalable distributed applications,” in Proc. International Conference on Autonomic Computing, pp. 132-139, May 17-18, 2004.

24. T. De Wolf, G. Samaey, T. Holvoet, and D. Roose, “Decentralised autonomic computing: Analysing self-organising emergent behaviour using advanced nu-merical methods,” in Proc. Second International Conference on Autonomic Computing ICAC 2005, pp. 52-63, June 13-16, 2005.

25. O. Babaoglu, M. Jelasity, and A. Montresor, Unconventional Programming Paradigms, vol. 3566/2005 of Lecture Notes in Computer Science, ch. Grassroots Approach to Self-management in Large-Scale Distributed Systems, pp. 286-296. Springer Berlin / Heidelberg, August 2005.

26. G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart, and S. R. White, “A multi-agent systems approach to autonomic computing,” in AAMAS ’04: Proceedings of the Third International Joint

Conference on Autonomous Agents and Multiagent Systems, (Washington, DC, USA), pp. 464-471, IEEE Computer Society, 2004.

27. D. Bonino, A. Bosca, and F. Corno, “An agent based autonomic semantic platform,” in Proc. International Conference on Autonomic Computing, pp. 189-196, May 17-18, 2004.

28. G. Kaiser, J. Parekh, P. Gross, and G. Valetto, “Kinesthetics extreme: an external infrastructure for monitoring distributed legacy systems,” in Proc. Autonomic Computing Workshop, pp. 22-30, June 25, 2003.

29. C. Karamanolis, M. Karlsson, and X. Zhu, “Designing controllable computer systems,” in HOTOS’05: Proceedings of the 10th conference on Hot Topics in Operating Systems, (Berkeley, CA, USA), pp. 49-54, USENIX Association, 2005.

30. G. Valetto, G. Kaiser, and D. Phung, “A uniform programming abstraction for effecting autonomic adaptations onto software systems,” in Proc. Second International Conference on Autonomic Computing ICAC 2005, pp. 286-297, June 13-16, 2005.

31. M. M. Fuad and M. J. Oudshoorn, “An autonomic architecture for legacy systems,” in Proc. Third IEEE International Workshop on Engineering of Autonomic and Autonomous Systems EASe 2006, pp. 79-88, Mar. 27-30,2006.

32. E. Bruneton, T. Coupaye, and J.-B. Stefani, “The fractal component model,” tech. rep., France Telecom R&D and INRIA, Feb. 5 2004.

33. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, “Chord: a scalable peer-to-peer lookup protocol for internet applications,” IEEE/ACM Transactions on Networking, vol. 11, pp. 17-32, Feb. 2003.

34. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, “A scalable content-addressable network,” in SIGCOMM ’01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, (New York, NY, USA), pp. 161-172, ACM, 2001.

35. I. T. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” in Middleware ’01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, (London, UK), pp. 329-350, Springer-Verlag, 2001.

36. M. Ohara, P. Nagpurkar, Y. Ueda, and K. Ishizaki, “The data-centricity of web 2.0 workloads and its impact on server performance,” in ISPASS, pp. 133-142, IEEE, 2009.

37. R. Ramakrishnan and J. Gehrke, Database Management Systems. Berkeley, CA, USA: Osborne/McGraw-Hill, 2nd ed., 2000.

38. O. Tickoo, R. Iyer, R. Illikkal, and D. Newell, “Modeling virtual machine per-formance: challenges and approaches,” SIGMETRICS Perform. Eval. Rev., vol. 37, pp. 55-60, Jan. 2010.

39. R. Iyer, R. Illikkal, O. Tickoo, L. Zhao, P. Apparao, and D. Newell, “VM3: Measuring, modeling and managing VM shared resources,” Computer Networks, vol. 53, pp. 2873-2887, December 2009.

40. M. Arlitt and T. Jin, “A workload characterization study of the 1998 world cup web site,” Network, IEEE, vol. 14, pp. 30 -37, may/jun 2000.

41. C. Lu, G. A. Alvarez, and J. Wilkes, “Aqueduct: Online data migration with performance guarantees,” in Proceedings of the 1st USENIX Conference on File and Storage Technologies, FAST ’02, (Berkeley, CA, USA), USENIX Association, 2002.

42. H. C. Lim, S. Babu, and J. S. Chase, “Automated control for elastic storage,” in Proceedings of the 7th international conference on Autonomic computing, ICAC ’10, (New York, NY, USA), pp. 1-10, ACM, 2010.

43. M. Parashar and S. Hariri, “Autonomic computing: An overview,” in Uncon-ventional Programming Paradigms, pp. 257-269, 2005.

44. Al-Shishtawy, V. Vlassov, P. Brand, and S. Haridi, “A design methodology for self-management in distributed environments,” in Computational Science and Engineering, 2009. CSE ’09. IEEE International Conference on, vol. 1, (Vancouver, BC, Canada), pp. 430-436, IEEE Computer Society, August 2009.

45. R. Das, J. O. Kephart, C. Lefurgy, G. Tesauro, D. W. Levine, and H. Chan, “Autonomic multi-agent management of power and performance in data centers,” in AAMAS ’08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems, (Richland, SC), pp. 107-114, International Foundation for Autonomous Agents and Multiagent Systems, 2008.

46. J. Kephart, H. Chan, R. Das, D. Levine, G. Tesauro, F. Rawson, and C. Lefurgy, “Coordinating multiple autonomic managers to achieve specified power- performance tradeoffs,” in Autonomic Computing, 2007. ICAC ’07. Fourth International Conference on, pp. 24-24, June 2007.

47. S. Bouchenak, F. Boyer, S. Krakowiak, D. Hagimont, A. Mos, J.-B. Stefani, N. de Palma, and V. Quema, “Architecture-based autonomous repair management: An application to J2EE clusters,” in SRDS ’05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems, (Orlando, Florida), pp. 13-24, IEEE, Oct. 2005.

48. J. O. Kephart and R. Das, “Achieving self-management via utility functions,” IEEE Internet Computing, vol. 11, no. 1, pp. 40-48, 2007.

49. S. Abdelwahed and N. Kandasamy, “A control-based approach to autonomic performance management in computing systems,” in Autonomic Computing: Concepts, Infrastructure, and Applications (M. Parashar and S. Hariri, eds.), ch. 8, pp. 149-168, CRC Press, 2006.

50. V. Bhat, M. Parashar, M. Khandekar, N. Kandasamy, and S. Klasky, “A self-managing wide-area data streaming service using model-based online control,” in Grid Computing, 7th IEEE/ACM International Conference on, pp. 176— 183, Sept. 2006.

51. R. Yanggratoke, F. Wuhib, and R. Stadler, “Gossip-based resource allocation for green computing in large clouds,” in Network and Service Management (CNSM), 2011 7th International Conference on, pp. 1 -9, oct. 2011.

52. Ali-Eldin, J. Tordsson, and E. Elmroth, “An adaptive hybrid elasticity controller for cloud infrastructures,” in The 13th IEEE/IFIP Network Operations and Management Symposium, NOMS 2012, (Hawaii, USA), April 2012.

метадані

Заголовок

Імплементація моделей покращення керованості великих масштабованих розподілених обчислювальних систем

Автор

Куцела Я.Т. Науковий керівник / Експерт

підрозділ

King Danylo University

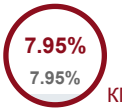
Тривога

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.

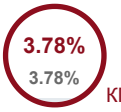
Заміна букв	ß	0
Інтервали	A→	0
Мікропробіли	·	0
Білі знаки	ß	0
Парафрази (SmartMarks)	a	73

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

17520
Кількість слів

137958
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/123456789/391/%D0%9F%D0%B0%D1%85%D0%BE%D0%BB%D1%8C%D1%87%D1%83%D0%BA%20%D0%9E.%D0%A0.%20%D0%B4%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0.pdf?sequence=1	57	0.33 %
2	https://alshishtawy.github.io/pdfs/publications/CAC2013_ElastMan.pdf	45	0.26 %
3	https://alshishtawy.github.io/pdfs/publications/CAC2013_ElastMan.pdf	41	0.23 %
4	http://repository.ukd.edu.ua/bitstream/handle/123456789/391/%D0%9F%D0%B0%D1%85%D0%BE%D0%BB%D1%8C%D1%87%D1%83%D0%BA%20%D0%9E.%D0%A0.%20%D0%B4%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0.pdf?sequence=1	39	0.22 %
5	http://people.cs.vt.edu/~irchen/5984/pdf/Nitti-TKDE14.pdf	39	0.22 %