БАКАЛАВРСЬКА РОБОТА

БР.ІП – 16.00.00.000 ПЗ

Група ІП-22-1

Смачинська Соломія

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Смачинська Соломія Орестівна

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Розробка fault-tolerant систем у розподілених середовищах
(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

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

Здобувач освітнього ступеня _____ С.О. Смачинська
(підпис, ініціали та прізвище здобувача)

Науковий керівник ____Піх Володимир Ярославович, к.т.н., доцент_____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

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

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

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

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Смачинській Соломії Орестівні

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Розробка fault-tolerant систем у розподілених середовищах"

керівник проекту (роботи) Піх Володимир Ярославович, к.т.н., доцент.

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

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

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

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

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

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

1 Різні типи несправностей (апаратні, програмні, мережеві та екологічні) (рис. 1.1, ст. 12)

2 Загальна конструкція FADI (рис. 2.1, ст. 31)

3 Загальна архітектура системи (рис. 2.3, ст. 40)

4 Розподілений застосунок з використанням FRIENDS (рис. 2.4, ст. 41)

5 Класи метаоб'єктів відмовостійкості (рис. 3.2, ст. 65)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання **2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	17.01.2026	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	21.02.2026	виконано
3	Побудова моделі або алгоритму власного рішення	01.03.2026	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	21.04.2026	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	02.05.2026	виконано

Студент _____Смачинська С.О.
(підпис)

Керівник роботи _____Піх В.Я.
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 89 сторінок, 24 рисунків, 4 таблиці, список використаних джерел із 40 найменувань.

Метою роботи є дослідження принципів та методів розробки fault-tolerant систем у розподілених середовищах.

Об'єкт дослідження: розподілені програмні системи.

Предмет дослідження: методи, моделі та інструменти забезпечення відмовостійкості у розподілених середовищах, зокрема механізми реплікації, балансування навантаження, виявлення та обробки збоїв.

Результати дослідження: проведено аналіз основних принципів побудови fault-tolerant систем, досліджено причини виникнення відмов у розподілених середовищах, розглянуто сучасні технології забезпечення відмовостійкості та оцінено ефективність їх використання.

У першому розділі розглянуто теоретичні основи розробки відмовостійких розподілених систем, їх основні властивості, принципи проєктування та комунікаційні моделі.

У другому розділі досліджено методи та інструменти забезпечення відмовостійкості, включаючи класифікацію відмов, методи їх виявлення та відновлення, а також технології забезпечення високої доступності.

У третьому розділі розглянуто практичні аспекти реалізації fault-tolerant рішень, архітектурні підходи до побудови систем та проведено оцінювання ефективності запропонованих методів.

Висновок: застосування сучасних підходів до забезпечення відмовостійкості дозволяє підвищити надійність і доступність розподілених систем, забезпечити безперервність їх роботи та ефективно обробляти збої в умовах динамічного середовища.

КЛЮЧОВІ СЛОВА: FAULT-TOLERANCE, РОЗПОДІЛЕНІ СИСТЕМИ, ВІДМОВОСТІЙКІСТЬ, РЕПЛІКАЦІЯ, БАЛАНСУВАННЯ НАВАНТАЖЕННЯ, ВИЯВЛЕННЯ ЗБОЇВ, ВИСОКА ДОСТУПНІСТЬ.

ANNOTATION

The bachelor's thesis contains of 89 pages, 24 figures, 4 tables, and a reference list with 40 sources.

The aim of the work is to study the principles and methods of developing fault-tolerant systems in distributed environments and to analyze approaches to building reliable, scalable, and resilient software systems.

Research object: distributed software systems.

Research subject: methods, models, and tools for ensuring fault tolerance in distributed environments, including replication mechanisms, load balancing, and fault detection and recovery techniques.

Research results: the main principles of building fault-tolerant systems were analyzed, the causes of failures in distributed environments were studied, modern fault tolerance technologies were reviewed, and their effectiveness was evaluated.

The first section describes the theoretical foundations of fault-tolerant distributed systems, their key properties, design principles, and communication models.

The second section analyzes methods and tools for ensuring fault tolerance, including classification of failures, detection and recovery techniques, and technologies for achieving high availability.

The third section covers the implementation aspects of fault-tolerant solutions, architectural approaches to distributed systems, and evaluation of their effectiveness.

Conclusion: the use of modern fault tolerance approaches improves the reliability and availability of distributed systems, ensures continuous operation, and enables effective handling of failures in dynamic environments.

KEY WORDS: FAULT TOLERANCE, DISTRIBUTED SYSTEMS, RELIABILITY, REPLICATION, LOAD BALANCING, FAILURE DETECTION, HIGH AVAILABILITY.

ЗМІСТ

ВСТУП

У ході виконання бакалаврської роботи було досліджено та обґрунтовано підходи до розробки fault-tolerant систем у розподілених середовищах, що є комплексним процесом, спрямованим на забезпечення високої надійності, доступності та стійкості програмних рішень до відмов. У роботі проаналізовано сучасні методи та технології побудови розподілених систем, розглянуто причини виникнення збоїв, а також досліджено ефективні механізми їх виявлення, обробки та запобігання. Особливу увагу приділено принципам побудови fault-tolerant архітектур, що відповідають вимогам сучасних високонавантажених систем. Процес дослідження охопив аналіз теоретичних основ, вибір інструментів, моделювання архітектурних рішень та оцінку їх ефективності.

Актуальність дослідження - в умовах стрімкого розвитку хмарних технологій, мікросервісної архітектури та розподілених обчислень забезпечення надійності програмних систем набуває особливої важливості. Сучасні інформаційні системи повинні безперервно функціонувати навіть за наявності апаратних, програмних або мережевих збоїв, оскільки від їхньої стабільності залежить якість надання послуг, безпека даних та задоволеність користувачів. Розробка fault-tolerant систем дозволяє мінімізувати ризики простоїв, забезпечити високу доступність сервісів і підтримувати узгодженість даних у розподіленому середовищі. Тому дослідження методів, технологій і підходів до створення відмовостійких розподілених систем є актуальним завданням сучасної інженерії програмного забезпечення.

Метою роботи є дослідження принципів та методів розробки fault-tolerant систем у розподілених середовищах.

Об’єкт дослідження: розподілені програмні системи.

Предмет дослідження: методи, моделі та інструменти забезпечення відмовостійкості у розподілених середовищах, зокрема механізми реплікації, балансування навантаження, виявлення та обробки збоїв.

Для досягнення поставленої мети у роботі використано такі методи дослідження: аналіз і узагальнення наукової, системний, порівняльний аналіз, методи аналізу, моніторингу, моделювання та проєктування програмних систем,

На початковому етапі було розглянуто основні поняття та компоненти надійних розподілених систем, включаючи такі характеристики, як відмовостійкість, масштабованість, доступність та узгодженість даних. Визначено ключові принципи проєктування, серед яких надлишковість, ізоляція компонентів, автоматичне відновлення та балансування навантаження. У другому розділі було досліджено причини виникнення відмов у розподілених системах, зокрема апаратні, програмні та мережеві збої. Розглянуто методи виявлення та діагностики помилок, включаючи моніторинг, логування, трасування та використання спеціалізованих інструментів спостереження. увагу приділено підходам до обробки збоїв, таким як повторні спроби, резервування ресурсів, реплікація даних, використання шаблонів Circuit Breaker та Bulkhead, що дозволяють підвищити стійкість системи до відмов. У третьому розділі розглянуто практичні аспекти реалізації fault-tolerant систем, включаючи використання платформних технологій та багаторівневих архітектур. Описано підходи до організації взаємодії між компонентами системи, зокрема через мікросервісну архітектуру та асинхронні комунікації. Проведено оцінку ефективності застосованих методів, що дозволило визначити їхній вплив на продуктивність, надійність та масштабованість системи.

Загалом, результати роботи підтверджують, що застосування сучасних методів забезпечення відмовостійкості є необхідною умовою для створення ефективних розподілених програмних систем. Перевагами запропонованих підходів є підвищення доступності сервісів, зменшення впливу збоїв на користувачів та можливість гнучкого масштабування системи. Водночас існують певні обмеження, зокрема збільшення складності архітектури та необхідність додаткових ресурсів для реалізації механізмів відмовостійкості.

Бакалаврська робота містить 89 сторінок, 24 рисунки, 4 таблиці, 3 розділи, список використаних джерел із 40 найменувань.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ ВІДМОВОСТІЙКИХ РОЗПОДІЛЕНИХ СИСТЕМ

1.1 Основні поняття та властивості fault-tolerant розподілених систем

Відмовостійкість є критичним аспектом розподілених систем, що забезпечує їхню безперервну роботу, незважаючи на виникнення несправностей. Відмовостійка система – це система, яка може переносити певні типи збоїв, не зазнаючи повного збою системи або втрати здатності виконувати важливі завдання. У розподілених системах несправності можуть виникати з різних джерел, включаючи апаратні збої, програмні помилки, перебої в мережі та людські помилки [22-31].

Типи несправностей у розподілених системах:

Збої обладнання: збої фізичних компонентів, таких як сервери, пристрої зберігання даних або мережева інфраструктура.

Збої програмного забезпечення: помилки, збої або неочікувані проблеми з поведінкою програмних застосунків, що працюють на розподіленій системі.

Збої в мережі: проблеми, такі як перевантаження, втрата повідомлень або розділення вузлів у мережі. Збої в навколишньому середовищі: зовнішні фактори, такі як відключення електроенергії, екстремальні температури або фізичне пошкодження інфраструктури.

Відмовостійкість у розподілених системах зазвичай досягається за допомогою кількох ключових механізмів: Надмірність та реплікація: ці механізми передбачають створення резервних копій даних або системних компонентів, щоб гарантувати, що у разі збою одного з них інший зможе взяти на себе його роботу [1]. Наприклад, дані можуть бути репліковані на кілька серверів або в кількох місцях розташування.

Контрольні точки: регулярне збереження станів системи, щоб забезпечити відновлення з відомих справних точок у разі збою.

Виявлення помилок: такі методи, як контрольні суми та системи моніторингу, для виявлення збоїв у міру їх виникнення.

Відновлення після помилок: Механізми відновлення після виявлених помилок шляхом перенаправлення завдань або перерозподілу ресурсів.

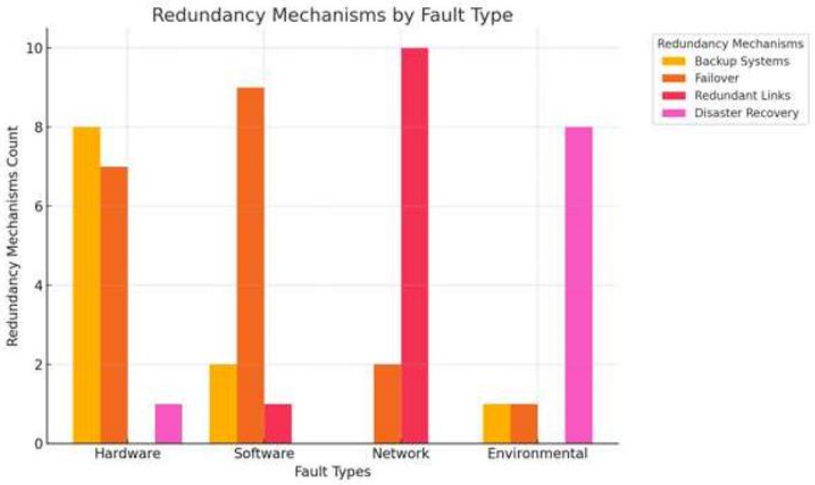

Рисунок 1.1 - Різні типи несправностей (апаратні, програмні, мережеві та екологічні).

Кожна група стовпчиків відповідає типу несправності, причому стовпчики всередині групи представляють різні механізми резервування.

Відмовостійкі алгоритми: Щоб забезпечити безперервне функціонування системи, незважаючи на збої, було розроблено різні алгоритми:

Paxos та Raft: Алгоритми консенсусу, що забезпечують узгодженість між розподіленими вузлами, навіть за наявності збоїв.

Підходи на основі кворуму: ці методи гарантують, що навіть якщо деякі вузли вийдуть з ладу, більшість (або кворум) вузлів все одно зможуть досягти згоди щодо даних або стану таблиця 1.1 .

Обмеження реального часу в критичних системах

Системи реального часу – це ті, які повинні реагувати на події або обробляти дані в рамках суворих часових обмежень. Ці обмеження зазвичай класифікуються на дві категорії:

Системи жорсткого реального часу: У цих системах пропуск терміну може призвести до катастрофічних наслідків. Наприклад, в аерокосмічній системі керування пропуск терміну може призвести до зриву місії або втрати життя.

Таблиця 1.1

Наведено порівняльне представлення ключових особливостей, переваг, проблем та ідеальних сценаріїв для кожного алгоритму.

Алгоритм	**Опис**	**Переваги**	**Виклики**	**Ідеальний сценарій використання**
Paxos	Алгоритм консенсусу для відмовостійкого узгодження.	- Перевірений та безпечний. - Строга узгодженість.	- Складна реалізація. - Кілька раундів обміну повідомленнями.	- Розподілені бази даних, банківські системи.
Raft	Спрощений алгоритм консенсусу.	- Легший для розуміння. - Вибір лідера.	- Накладні витрати на зв'язок. - Проблеми з відмовостійкістю в екстремальних випадках.	- Сховища типу «ключ-значення», мікросервіси.
Quorum-based	Досягає консенсусу через згоду більшості.	- Простий. - Масштабований та відмовостійкий.	- Падіння продуктивності при затримках. - Мережеві проблеми.	- Великомасштабні NoSQL бази даних (Cassandra тощо).

М’які системи реального часу: ці системи можуть терпіти випадкові затримки, але продуктивність знижується, якщо терміни часто пропускаються. Прикладом може бути потокове передавання мультимедіа, де випадкова буферизація є прийнятною, але часті затримки призводять до низької якості.

Щоб системи реального часу були ефективними, вони повинні надійно дотримуватися термінів, а також забезпечувати відмово стійкість [2]. У критичних системах застосовуються такі додаткові міркування:

Висока доступність: Система повинна бути завжди доступною для роботи, особливо в критично важливих для безпеки застосуваннях.

Безпека: Системи реального часу в критично важливих секторах, таких як охорона здоров'я чи транспорт, повинні гарантувати, що вони не завдають шкоди користувачам чи навколишньому середовищу. Наприклад, збій у системі керування автономним транспортним засобом може призвести до аварій.

Часові обмеження в відмовостійких системах:

Пропуски термінів: Стратегії відмовостійкості повинні бути розроблені

таким чином, щоб мінімізувати ймовірність пропуску термінів навіть у разі виникнення помилок.

Затримка: Механізми відмовостійкості не повинні призводити до значних затримок, оскільки це може поставити під загрозу виконання вимог щодо часу.

Передбачуваність: Система повинна демонструвати передбачувану поведінку в умовах відмови, що дозволяє інженерам проектувати системи з гарантованим часом виконання в найгіршому випадку.

Попередні дослідження з відмовостійких розподілених обчислень. Галузь відмовостійких розподілених обчислень значно розвинулася за останні кілька десятиліть, причому значна частина досліджень зосереджена на проектуванні систем, які зберігають функціональність навіть в умовах збоїв.

Таблиця 1.2

Показано, як кожна система підходить до забезпечення відмовостійкості, продуктивності в режимі реального часу та для яких областей вона найкраще підходить.

Система	**Стратегія відмовостійкості**	**Продуктивність у реальному часі**	**Сфери застосування**
Google Spanner	Синхронна реплікація, консенсус Paxos.	Висока пропускна здатність, низька затримка.	Хмарні бази даних, фінансові додатки.
Apache Kafka	Реплікація, вибір лідера, на основі логів.	Низька затримка, висока пропускна здатність.	Аналітика в реальному часі, системи подій.
Hazelcast	Сітка в пам'яті (In-memory grid), синхронна реплікація.	Обробка в реальному часі з низькою затримкою.	IoT, фінансові послуги.
Consul	Вибір лідера, перевірки стану (health checks).	Швидке виявлення сервісів, відновлення після збоїв.	Мікросервіси, SOA.
Cassandra	На основі кворуму, настроювана узгодженість.	Висока пропускна здатність запису, низька затримка.	Big data, електронна комерція.

Ключові віхи в цій галузі включають:

Методи реплікації: Ранні роботи над відмовостійкими розподіленими системами були зосереджені на реплікації критично важливих даних або служб для запобігання єдиній точці відмови [3]. Ця робота призвела до розробки таких

систем, як Bigtable від Google, яка використовує реплікацію для підтримки високої доступності.Алгоритми консенсусу: Розробка алгоритмів консенсусу, таких як Paxos та Raft, революціонізувала те, як розподілені системи обробляють відмовостійкість, гарантуючи, що більшість вузлів у системі погоджуються щодо стану, навіть якщо деякі вузли виходять з ладу таблиця 1.2.

Розподілені бази даних: Дослідження розподілених баз даних, включаючи такі методи, як реплікація типу "головний-підлеглий", обрання лідера та підходи на основі кворуму, сприяли підвищенню відмовостійкості великомасштабних систем.

Відмовостійкість у реальному часі: Дослідження, що стосуються систем реального часу в критично важливих додатках, зосереджені на забезпеченні того, щоб механізми відмовостійкості не заважали виконанню суворих вимог до часу [4]. Для систем реального часу були запропоновані такі методи, як планування на основі пріоритетів та алгоритми відмовостійкого планування.

Останні досягнення: Останні досягнення зосереджені на інтеграції нових технологій, таких як машинне навчання для прогнозного обслуговування, блокчейн для безпечних відмовостійких систем та периферійні обчислення для підвищення відмовостійкості на периферії мережі. Крім того, досягнення в механізмах самовідновлення та автономного відновлення після збоїв є перспективними для наступного покоління відмовостійких систем реального часу.

Еволюція відмовостійких систем відзначилася постійним удосконаленням резервування, реплікації, консенсусних алгоритмів та механізмів виявлення помилок. Однак проблема підтримки як відмовостійкості, так і продуктивності в режимі реального часу в критично важливих системах залишається важливим напрямком досліджень. Оскільки критично важливі програми стають більш складними та взаємопов'язаними, майбутні дослідження, ймовірно, будуть зосереджені на підвищенні масштабованості відмовостійких механізмів, забезпеченні гарантій роботи в режимі реального часу за різних умов відмови та інтеграції нових технологій, таких як штучний інтелект та квантові обчислення, для виявлення та відновлення після несправностей [5]. Цей

підрозділ закладає основу для розуміння фундаментальних принципів відмовостійких розподілених обчислень, проблем систем реального часу та поточних дослідницьких зусиль, спрямованих на підвищення як надійності, так і продуктивності в критично важливих середовищах. Надані графіки, таблиці та зображення допоможуть візуалізувати порівняння різних відмовостійких підходів та їх вплив на продуктивність системи.

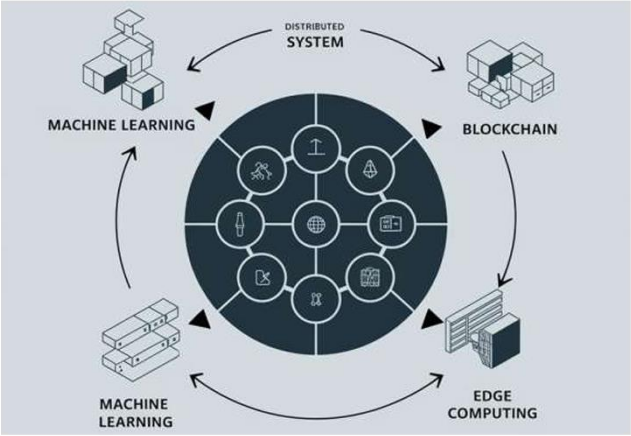

Рисунок 1.2 - На діаграмі показано, як новітні технології (наприклад, машинне навчання, блокчейн, периферійні обчислення) інтегруються з традиційними відмовостійкими розподіленими системами.

1.2 Принципи проєктування відмовостійких програмних систем

Простота використання. Механізми, що реалізують відмовостійкість та безпечний зв'язок, повинні бути легко використовуваними розробниками додатків [6]. Це також має бути вірним для самих розробників механізмів, для яких, наприклад, механізми групового зв'язку повинні бути простими у використанні. Коли в різних додатках використовуються різні механізми, програміст або користувач даного додатку повинен мати можливість вибрати адекватний механізм, навіть якщо його реалізація залишається прихованою. Таким чином, ми розрізняємо прозорість з точки зору використання та видимість з точки зору конфігурації.

Повторне використання. З точки зору розробника механізму, повторне використання може мати одну з двох наступних форм: *узагальненість* та

розширюваність. Узагальненість означає, що механізми можна повторно використовувати без модифікацій у нових за стосунках [7]. Розширюваність означає, що частину існуючого механізму необхідно оновлювати та модифікувати, щоб отримати механізм, що відповідає різним вимогам. Об'єктно-орієнтовані методи та мови проектування забезпечують хороший спосіб отримання обох властивостей: абстракція та інкапсуляція допомагають забезпечити узагальненість, тоді як успадкування або делегування дозволяють поступову розробку та розширюваність.

Компонування. Ця властивість стосується можливості використання кількох незалежних механізмів в одній програмі залежно від характеристик та вимог програми. Наприклад, кілька стратегій відмовостійкості можуть використовуватися для різних об'єктів в одній програмі без будь-яких конфліктів. Механізми також можуть бути різних типів: програмі можуть знадобитися як механізми відмовостійкості, так і механізми безпеки. Механізми безпеки можна досить легко вставляти або видаляти з програми без будь-якої зміни її вихідного коду. Визначення компонування наведено тут. подібно до поняття композиції протоколів надійності. Тим не менш, наше поняття є більш обмеженим; поняття композиції протоколів надійності розширює наше поняття компонуємості в тому сенсі, що заданий протокол відмовостійкості може бути складений з кількох елементарних протоколів.

Підхід, представлений у цій роботі, є розвитком попередніх робіт з інтеграції механізмів відмовостійкості в розподілених системах, або на системному рівні через системні служби, або на рівні користувача через використання бібліотек.

Системний *підхід* забезпечує механізми відмовостійкості, вбудовані в базову систему виконання, які є (майже) прозорими для програміста додатків. Наприклад, система Delta-4 пропонує кілька протоколів реплікації, заснованих на системі багатоадресного зв'язку, що підтримує протоколи виявлення помилок та голосування. У цьому випадку механізми прозорі та прості у використанні. Однак, оскільки вони інтегровані в операційну систему, до механізмів нелегко отримати доступ та налаштувати їх . Крім того, компонування різних механізмів,

що працюють з різними класами несправностей, не завжди можливе. Додавання різних типів механізмів до додатка в кожному окремому випадку, наприклад, автентифікація клієнт-сервер, є непростим.

Бібліотечний *підхід* надає базові механізми, що дозволяють користувачам налаштовувати власні механізми відмовостійкості відповідно до їхніх потреб за допомогою конструкцій та примітивів. Наприклад, Isis [8] пропонує програмні конструкції (наприклад, *координатор-когорта),* управління групами та атомарні широкомовні примітиви, на основі яких, наприклад, можна побудувати первинну/резервну та активну реплікацію. Хоча бібліотечний підхід є більш гнучким, він, однак, відповідає за правильне використання бібліотечних функцій у відповідних місцях вихідного коду для реалізації заданого механізму відмовостійкості. Ця здатність може вимагати добрих знань методів відмовостійкості та, по суті, не має простоти використання.

Об'єктно -*орієнтована розробка* таких бібліотек надає користувачеві класи, а не функції. Таким чином, успадкування спрощує поступову адаптацію механізмів відмовостійкості до конкретних потреб або додавання нових функцій. Таким чином, цей підхід повинен досягати дуже хорошої можливості повторного використання. Приклади такого використання успадкування можна знайти в Avalon/C++ та Arjuna [9]. Отже, з одного боку, системний підхід забезпечує повну прозорість механізмів, а з іншого боку, об'єктно-орієнтовані бібліотеки забезпечують можливість повторного використання, але жоден з цих підходів не вдається поєднати ні властивості, ні... забезпечує задовільне рішення щодо компонуємості. Фактично, уважне спостереження за кодом, написаним користувачами бібліотеки, показує, що функції використовуються майже систематично в певних точках обчислювальної моделі, таких як створення та видалення об'єктів, початок та завершення методів. Це саме та проблема, яку протокол метаоб'єктів може вирішити елегантним способом.

Суть метаоб'єктних протоколів (МОР) полягає в тому, щоб надати користувачам можливість налаштовувати реалізацію мови відповідно до їхніх конкретних потреб [10]. Метаоб'єктні протоколи базуються на рефлексії та об'єктно-орієнтованій системі. Рефлексія надає реалізацію мови на високому

рівні абстракції, роблячи її зрозумілою для користувача, зберігаючи при цьому ефективність та портативність реалізації мови за замовчуванням. Об'єктно-орієнтована система надає інтерфейс до реалізації мови у вигляді класів та методів, щоб можна було створювати варіанти реалізації мови за замовчуванням, використовуючи спеціалізацію шляхом успадкування. Екземпляри таких класів називаються *метаоб'єктами.* Поняття *протоколу* тут стосується взаємодії між об'єктом та метаоб'єктом. У рефлексивних мовах на основі класів цей інтерфейс зазвичай включає щонайменше створення та видалення екземплярів, доступ для читання або запису атрибутів , виклик методу. Апріорним аргументом проти мов на основі рефлексії є те, що вони неефективні. Контрприкладом є ABCL/R2, об'єктно-орієнтована паралельна рефлексивна мова, яка за продуктивністю порівнюється з C, що використовується з легкими процесами [11]. Рефлексія під час компіляції призводить до хорошої продуктивності щодо часткової оцінки.

Можливість адаптувати деякі аспекти мовної реалізації може бути делегована третій стороні, а не користувачеві, наприклад, спеціалісту з відмовостійкості або безпеки. Таким чином, можна досягти чіткого розмежування обов'язків між застосунком та механізмами відмовостійкості або безпечного зв'язку. Такий підхід дозволяє чітко визначити роль різних програмістів з різними базовими знаннями та спростити їхнє завдання: програміст застосунку, програміст відмовостійкості, програміст безпеки, програміст дистрибутиву. Усі вони мають однакові знання об'єктно-орієнтованого програмування та деякі знання про метаоб'єктні протоколи.

У простому протоколі метаоб'єктів метаоб'єкт планується для виконання певних дій, коли об'єкт створюється (видаляється) або коли викликається метод об'єкта. І навпаки, метаоб'єкт може активувати методи об'єкта та отримувати доступ до атрибутів об'єкта (стану об'єкта). На практиці взаємодія між клієнтським об'єктом та серверним об'єктом може контролюватися кількома метаоб'єктами.

Цей підхід можна застосовувати рекурсивно, щоб додавання деяких механізмів, пов'язаних з надійністю, відбувалося простим та систематичним способом. Тим не менш, протоколи метаоб'єктів не є панацеєю, і тут не

стверджується, що їх можна використовувати самостійно для побудови надійних розподілених систем. На системному рівні необхідно реалізувати кілька базових служб, таких як виявлення помилок (для забезпечення високого охоплення припущень про режим відмови) або ядро безпеки (яке має завжди викликатися та бути захищеним від несанкціонованого доступу). Також необхідні інші системні служби, такі як протоколи управління групами та атомарної багатоадресної розсилки, сервери автентифікації та авторизації .

Система FRIENDS поділяє ту саму філософію, що й тісно пов'язані проекти, такі як M **AUD** та G **ARF**. Ідея полягає в тому, щоб обробляти механізми надійності на окремому рівні абстракції та прив'язувати їх до об'єктів програми відповідно до їхніх потреб. У вищезгаданих прикладах підхід спирається головним чином на здатність перехоплювати виклики та перенаправляти їх на певний поведінковий об'єкт або метаоб'єкт. Перехоплення більше спирається на деякі можливості системи виконання для перенаправлення повідомлень, ніж на мовні засоби. Також створення об'єктів матеріалізується системою виконання, а не мовою [12]. Наш підхід більше спирається на мовні засоби для обробки створення об'єктів, викликів , а також стану (атрибутів) об'єкта. Реифікація об'єкта , як поведінки, так і інформації про стан, має велике значення з точки зору відмовостійкості . Чим потужніша рефлексивна мова, тим більше інформації про стан об'єкта можна оптимізувати в контрольних точках. Головною перевагою мовного підходу є те, що не потрібні припущення щодо базової системи. Головним недоліком є те, що потрібна специфічна мова, і що все ще необхідно дотримуватися деяких правил програмування.

У проекті використовують рефлексивні можливості мови та метаоб'єктів для обробки обмежень реального часу, відмовостійкості програмного забезпечення або транзакційних моделей [13]. Обробка проблем реального часу на метарівні є складнішою, оскільки вона передбачає доступ до функцій, які зазвичай призначені для внутрішніх компонентів ядра системи. Усі ці роботи по-різному ілюструють інтерес до рефлексії в надійних обчисленнях.

У нашій попередній роботі було досліджено використання метаоб'єктів для реалізації механізмів відмовостійкості з використанням одного метарівня.

Було розроблено та експериментально проведено кілька класів метаоб'єктів для різних стратегій реплікації. Це було дуже перспективним і показало, що для програміста додатків можна досягти прозорості та розділення завдань.

Однак, підхід на одному метарівні має кілька недоліків. По-перше , взаємодія між репліками, що обробляються на метарівні, була досить складною через використання системних викликів до служб управління групами. Аналогічно, віддалена взаємодія між об'єктами програми була реалізована на базовому рівні і, таким чином, дуже залежала від використовуваних механізмів зв'язку [14]. Другою проблемою була складність прозорого додавання деяких аспектів безпеки (автентифікація, шифрування та перевірка підписів). Усі оператори, пов'язані з безпекою, були змішані з вихідним кодом для відмовостійкості та управління групами. Оскільки віддалена взаємодія та безпека не оброблялися як окремі та незалежні абстракції, гнучкість та можливість повторного використання метаоб'єктів, які були розроблені спочатку, були дуже обмеженими.

Це було першою мотивацією для розділення в межах кількох метаоб'єктів відмовостійкості, безпеки та групового зв'язку. Ідея полягала в тому, щоб приховати розподіл на рівні відмовостійкості за допомогою метаоб'єктів зв'язку, що призвело б до двох метарівнів замість одного. Розділення питань відмовостійкості та протоколів розподілу як двох окремих метарівнів дозволяє вставляти механізми безпеки як новий метарівень.

1.3 Комунікаційні моделі та протоколи в розподілених середовищах

Переваги відмовостійкості зазвичай рекламуються як покращення надійності - рівня довіри, який можна обґрунтовано покласти на систему. Зазвичай надійність визначається статистичною термінологією, яка вказує на ймовірність того, що система функціональна та забезпечує очікуваний сервіс у певний момент часу. Це призводить до поширених визначень, таких як добре відомий *середній час до відмови* (MTTF). Хоча такі терміни, як надійність, безвідмовність та доступність, важливі в практичних умовах, вони не є

центральними для цієї роботи, оскільки ми зосереджуємося на етапі проектування відмовостійкості, а не на етапі оцінювання [15]. Тому, хоча вищезазначені терміни можуть залишатися дещо неточними, важливо точно визначити характеристики розподіленої системи. Це зроблено в наступному розділі. Для отримання точних визначень атрибутів надійності.

Ми моделюємо розподілену систему як скінченну множину процесів, які взаємодіють, надсилаючи повідомлення з фіксованого алфавіту повідомлень через комунікаційну підсистему. Змінні кожного процесу визначають його *локальний стан* . Кожен процес виконує локальний алгоритм, який призводить до послідовності атомарних переходів його локального стану. Кожен перехід стану визначає *подію* , яка може бути подією *надсилання* , подією *отримання* або *внутрішньою* подією. Жодних припущень щодо топології мережі або властивостей доставки повідомлень комунікаційної системи не робиться, окрім того, що надсилання повідомлень між довільними процесами має бути можливим.

Ми використовуємо *захищені команди* як позначення для абстрактного представлення локального алгоритму. Захищена команда - це пара, що складається з булевого виразу над локальним станом (який називається *guard*) та присвоєння, що означає атомарну та миттєву зміну локального стану (який називається *command).* Вона записується як *{guard) з {command*). Захищена команда (яку для різноманітності іноді називають *дією*) називається увімкненою, *якщо* її guard оцінюється як true. *Локальний алгоритм* - це набір захищених команд (у позначенні команди розділені символом '[]'). Процес виконує крок у своєму локальному алгоритмі, оцінюючи всі свої guard та недетерміновано вибираючи одну увімкнену дію, з якої він виконує присвоєння. Ми припускаємо, що вибір дій є справедливим (тобто дія, яка вмикається нескінченно часто, зрештою вибирається та виконується); це відповідає *сильній справедливості*. Зв'язок вбудований у нотацію наступним чином: Команда guard може містити спеціальний оператор rcv, який оцінюється як true, якщо відповідне повідомлення може бути отримано. Команда може містити один або декілька операторів snd, які, фактично, надсилають повідомлення іншому процесу.

Розподілена програма (або *розподілений алгоритм*) складається з набору локальних алгоритмів. Загальний системний стан такої програми, який називається конфігурацією , складається з усіх локальних станів усіх процесів плюс стан комунікаційної підсистеми (тобто повідомлень, що передаються). Щоб зробити цю роботу легшою для читання, часто використовуємо терміни конфігурація та стан як синоніми та явно пишемо «локальний» стан, коли йдеться про локальний стан процесу. Усі процеси мають локальний *початковий стан* , який визначає *початкову конфігурацію* системи. Ми не намагаємося точно розрізняти терміни розподілена система, розподілена програма та розподілений алгоритм. Хоча останні два використовуються як синоніми, перший зазвичай стосується всієї сукупності програми, стану та середовища виконання (тобто апаратного забезпечення).

Щоб краще зрозуміти всі ці визначення, розглянемо лістинг 1.1, на якому зображено простий розподілений алгоритм, що використовує нотацію захищених команд [16]. Алгоритм є добре відомим прикладом «пінг-понгу» двох процесів, які по черзі надсилають повідомлення один одному. Наприклад, локальний стан процесу *Ping* складається зі значень змінних *z* та *ack* (які ініціалізовані відповідно 0 та **true**). Глобальний стан, у свою чергу, складається зі значень усіх змінних (z, *ack* та *wait),* а також набору повідомлень, які можуть перебувати в дорозі (тобто *a* або *m*). Початкова конфігурація – це глобальний стан, де z = 0, *ack* = **правда** , *зачекайте* = **true** , і де жодних повідомлень не передається. Зверніть увагу, що в кожен момент часу в обох процесах завжди є щонайбільше одна увімкнена дія: або це черга процесу надсилати, або процес очікує на отримання. Якщо захищена команда увімкнена, її можна виконати, що призведе до зміни локального стану та так само до зміни глобальної конфігурації.

Лістинг 1.1 - Простий розподілений алгоритм.

```
process Ping
  var z : ℕ init 0
   ack : boolean init true
  begin
      ¬ack ∧ rcv(m)   ⟶   ack := true; z := z + 1
   ▯  ack             ⟶   snd(a); ack := false
  end

process Pong
  var wait : boolean init true
  begin
      ¬wait           ⟶   snd(m); wait := true
   ▯  wait ∧ rcv(a)   ⟶   wait := false
  end
```

У літературі існує значна неоднозначність щодо значення деяких центральних термінів, таких як помилка та збій. Зазначено, що «те, що одна людина називає збоєм, друга людина називає несправністю, а третя людина може назвати помилкою». Термін « *помилка »* зазвичай використовується для позначення дефекту на найнижчому рівні абстракції, наприклад, комірки пам'яті, яка завжди повертає значення 0. Несправність може спричинити помилку, яка є категорією стану системи. Помилка, фактично, може призвести до збою, тобто до відхилення системи від її специфікації правильності [17]. Ці терміни, слід визнати, є розпливчастими, і тут знову ж таки формалізація може допомогти уточненню.

Традиційно, несправності оброблялися шляхом опису результуючої поведінки системи та групувалися в ієрархічну структуру класів несправностей або *моделей несправностей*. Відомими прикладами є модель *аварійного* збою (за якої процесори просто припиняють виконуватися в певний момент часу), модель *зупинки внаслідок збою* (за якої процесор дає збій, але це може бути легко виявлено його сусідами) або *візантійська модель* (за якої процесори можуть поводитися довільно, навіть зловмисно). Коректність системи завжди доводилася стосовно конкретної моделі несправності. Але, на жаль, незначні неоднозначності або розбіжності у визначенні працювали проти будь-якого спільного розуміння навіть простих моделей несправностей, і тому були розроблені більш формальні методи для їх опису.

Формальний підхід до визначення терміна «збій» зазвичай ґрунтується на спостереженні, що системи змінюють свій стан в результаті двох досить схожих класів подій: нормальної роботи системи та виникнення збоїв. Таким чином, збій можна змоделювати як небажаний (але тим не менш можливий) перехід стану процесу. Використовуючи додаткові (віртуальні) змінні для розширення фактичного простору станів процесу, можна змоделювати будь-який тип збою із поширених класів збоїв [18]. Як приклад, розглянемо лістинг 1.2, на якому показано код одного з процесів на лістинг 1.1, який може бути схильний до збоїв у результаті аварії. Виникнення збою можна змоделювати, додавши внутрішню булеву «змінну збою» *до* стану процесу.

Лістинг 1.2 - Процес, який може призвести до збою.

```
process Pong that may crash
  var wait : boolean init true
  error up : boolean init true {* virtual error variable *}
  begin
      {* normal program actions: *}
   [] up ∧ ¬wait              ⟶  snd(m); wait := true
   [] up ∧ wait ∧ rcv(a)      ⟶  wait := false
      {* fault actions: *}
   [] up                      ⟶  up := false
  end
```

Тепер до набору захищених команд можна додати додаткову дію *up з up : = false, яка моделює збій. Щоб відповідати очікуваній поведінці моделі збою, усі інші дії процесу мають бути зупинені, чого можна досягти, додаючи* як додатковий кон'юнкт до захищених команд цих дій. Там, де віртуальні змінні помилок можуть перешкоджати нормальним діям програми, ми називаємо змінну помилки *ефективною. (Зауважте ,* що наступна дія виправлення повинна просто знову встановитися *в* значення true.) Додавання віртуальних змінних помилок та дій щодо несправностей можна розглядати як перетворення початкової програми на потенційно несправну програму. Дослідження та - покращення властивостей таких програм є предметом методологій

відмовостійкості.

Ми називаємо стан, що містить такі додаткові змінні помилки, *розширеним локальним станом* або *розширеною конфігурацією,* коли йдеться про всю систему. Тому ми визначаємо *помилку* як дію на можливо розширений стан процесу. Набір помилок називається *класом помилок*. Переваги цього визначення полягають не лише в ясності його семантики; визначаючи помилки таким чином, можна легко міркувати про програми, схильні до помилок, використовуючи стандартні методи, спрямовані на нормальну безвідмовну роботу. Ми уникаємо терміна *«помилка»* та використовуємо термін *«відмова»* для позначення того факту, що система не поводилася відповідно до своєї специфікації.

Ми приймаємо звичайні визначення властивостей системи, згідно з якими *виконання* розподіленої програми є нескінченною послідовністю $e = {}_{c0}, c_1, c2_{,}$. . . глобальних системних конфігурацій. Конфігурація *c0* є початковою конфігурацією, а всі наступні конфігурації $ct_{(}$ з $i > 0$) є результатом $c_{i} - {}_{1}$ шляхом виконання одного увімкненого захищеного оператора [19]. Кінцеві виконання - (наприклад, завершення програм) технічно перетворюються на нескінченні - послідовності шляхом нескінченного повторення кінцевої конфігурації. У місцях, де ми явно згадуємо кінцеві виконання, ми називаємо їх *частковими виконаннями* .

Властивість розподіленої програми – це набір системних виконань. Розподілена програма завжди визначає властивість сама по собі, яка є множиною всіх системних виконань, можливих з її початкової конфігурації. Кажуть, що певна властивість p має місце *для розподіленої* програми, якщо набір послідовностей, визначених програмою, міститься в p .

Два основні класи властивостей системи, необхідних для опису будь-якої корисної поведінки системи: безпека та життєздатність. Неформально, *властивість безпеки* стверджує, що якась конкретна «погана річ» ніколи не трапляється в системі. Це можна формально охарактеризувати, вказавши, коли виконання e є небезпечним для властивості p (тобто не міститься у властивості безпеки p): якщо $e \approx p$ *, то в межах* e має бути ідентифікована дискретна подія

, яка забороняє всім можливим продовженням виконання бути безпечними (це небажана та невиправна «погана річ»). Наприклад, розглянемо світлофор, який керує рухом на перехресті доріг [20]. Просту властивість безпеки цієї системи можна сформулювати так: У будь-який момент часу жодні два світлофори не повинні показувати зелений колір. Система не була б безпечною, якби існувало виконання системи, де два різні світлофори показують зелений колір.

Властивість безпеки зазвичай виражається набором «допустимих» конфігурацій системи, які зазвичай називають *інваріантними.* Доводячи, що розподілена програма є безпечною, ми гарантуємо, що система завжди залишатиметься в межах цього набору безпечних станів. Таким чином, властивість безпеки безумовно забороняє системі перемикатися в конфігурації, які не входять до цього набору. (Загалом, якщо порушення властивості p можна спостерігати за скінченний час, то p є властивістю безпеки.)

З іншого боку, *властивість живучості* стверджує, що якась «хороша річ» зрештою станеться під час виконання системи . Формально, часткове виконання системи є *живим* для властивості p тоді і тільки тоді, коли його можна розширити, щоб він все ще залишався в p . *Властивість живучості - це така, для якої кожне часткове виконання є живим. Розглянемо ще* раз приклад зі світлофором. Властивість живучості можна сформулювати так: автомобіль, що чекає на червоне світло світлофора, врешті-решт повинен отримати зелений сигнал і йому буде дозволено перетнути перехрестя. Це показує, що властивості живучості є властивостями «імовірності» (на відміну від властивостей «завжди», таких як безпека). Кожне часткове виконання має бути живим, тобто система гарантує, що водій автомобіля врешті-решт перетне вулицю, де відбудеться очікувана «хороша річ». Зауважте, що ця хороша річ не повинна бути дискретною: властивість живучості стосується перетину *всіх* автомобілів, які прибувають на перехрестя. Таким чином, властивості живучості можуть фіксувати поняття прогресу.

Найпоширенішим прикладом життєздатності в розподілених системах є *завершення.* Прикладами, коли задана «хороша річ» не є дискретною, є такі властивості, як *гарантоване обслуговування* (яке стверджує, що кожен запит

зрештою буде задоволено). Властивості життєздатності не прогнозують, коли «хороша річ» станеться, вони лише підтримують її можливість. Фактичний доказ того, що система задовольняє властивість життєздатності, зазвичай включає аргумент обґрунтованості.

Лістинг 1.3 - Проста відмовостійка програма.

```
process from Example 1
  var x ∈ {0, 1, 2, 3} init 1 {* local state *}
  begin
      {* normal program actions: *}
      x = 1   ⟶   x := 2
   ▯  x = 2   ⟶   x := 1
      {* faults that should be tolerated: *}
   ▯  true    ⟶   x := 0
      {* protection mechanism: *}
   ▯  x = 0   ⟶   x := 1
  end
```

Специфікація задачі складається з властивості безпеки та властивості живучості. Розподілений алгоритм *A* називається коректним щодо *специфікації задачі, якщо для A* виконуються як властивість безпеки, так і властивість живучості .

1.4 Висновки по розділу

У першому розділі було розглянуто теоретичні основи розробки fault-tolerant систем у розподілених середовищах. Проаналізовано основні поняття, характеристики та властивості відмовостійких розподілених систем, а також їх значення для забезпечення безперервної роботи програмного забезпечення. Досліджено принципи проєктування відмовостійких програмних систем, що дозволяють підвищити надійність, доступність та стійкість до збоїв. Також розглянуто комунікаційні моделі та протоколи, які використовуються у розподілених середовищах для організації взаємодії між компонентами системи. Отримані результати формують теоретичну основу для подальшого аналізу методів забезпечення відмовостійкості.

РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ ЗАБЕЗПЕЧЕННЯ ВІДМОВОСТІЙКОСТІ

2.1 Класифікація відмов та їх вплив на розподілені системи

Загальновизнано, що розподілені системи є основою сучасного комп'ютерного світу. Однією з очевидних переваг розподілених систем є їхня здатність ефективно вирішувати складні проблеми, що потребують величезної обчислювальної потужності. Виконання ресурсомістких програм, таких як навчання нейронних мереж або моделювання різних систем, може бути значно пришвидшено лише за рахунок використання паралелізму. Ще однією перевагою розподілених систем є те, що вони відображають глобальне бізнес- та соціальне середовище, в якому ми живемо та працюємо. Впровадження електронної комерції, систем бронювання авіаквитків, систем супутникового спостереження або систем телеметрії в реальному часі немислиме без послуг розподілених систем всередині країни та за її межами [21].

Розгортання розподілених систем у цих областях поставило додаткові вимоги до їхньої надійності. Для розподіленої обробки обчислювальних програм важливо впроваджувати заходи стійкості до відмов, щоб уникнути марнування обчислень, що виконуються на всій розподіленій системі, коли один з її вузлів виходить з ладу. В онлайн-системах та критично важливих системах збій у роботі системи може порушити роботу систем керування, зашкодити продажам або поставити під загрозу людське життя. Розподілені середовища, в яких працюють такі програми, повинні бути високодоступними, тобто вони повинні продовжувати надавати стабільні та точні послуги, незважаючи на несправності базового обладнання.

FADI була розроблена для використання обчислювальної потужності взаємопов'язаних робочих станцій для надання надійних розподілених обчислювальних послуг за наявності апаратних збоїв, що впливають на окремі вузли в розподіленій системі . Для досягнення поставленої мети FADI повинна підтримувати автоматичний розподіл прикладних процесів та забезпечувати

засоби для прозорої для користувача відмовостійкості в багатовузловому середовищі.

Вирішення проблем надійності розподілених систем передбачає вирішення двох проблем: виявлення помилок та відновлення процесів. Виявлення помилок стосується постійних та тимчасових несправностей комп'ютерного обладнання, а також несправностей у прикладному програмному забезпеченні та каналах зв'язку. Відновлення розподілених програм, що вийшли з ладу, вимагає відновлення локального стану виконання процесів, а також врахування стану каналів зв'язку між ними на момент збою

Огляд системи FADI

FADI розроблено для підтримки обчислювально ресурсоємних програм, що виконуються на мережевих робочих станціях. Хоча «економія зусиль» передбачає впровадження заходів, що запобігають втраті результатів тривалих обчислень, що виконуються на розподілених вузлах, часто важко виправдати використання дорогих методів відмовостійкості апаратної реплікації. Це часто трапляється з обчислювально ресурсоємними автономними програмами, такими як моделювання масо- та теплопередачі [1], візуалізація графіки [2] тощо. Ще однією актуальною областю застосування FADI є розподілені онлайн-програми, такі як механізми підтримки прийняття рішень [3] та системи бронювання авіаквитків [22]. Такі системи можуть терпіти коротку затримку в обслуговуванні – для управління несправностями, але повна зупинка системи при виникненні несправностей є неприйнятною. У цій роботі представлено економічно ефективне та ефективне відмовостійке рішення для запуску цих програм на мережевих системах.

На рисунку 2.1 представлено абстрактне уявлення про роботу FADI. Модулі *виявлення помилок* та *відновлення після збоїв* працюють на центральному вузлі (зазвичай на сервері), який вважається відмовостійким, тобто ймовірність його відмови незначна. Необхідна надійність центрального вузла може бути досягнута шляхом дублювання апаратного забезпечення, але деталі її досягнення не мають відношення до обговорення самого FADI. Для зберігання контрольних точок процесів також передбачається централізована файлова система NFS, яка

є видимою та ідентичною для всіх вузлів мережі.

Після запуску системи FADI визначає конфігурацію базової мережі та надає її користувачеві, який вибирає вузол(и) мережі, на яких мають виконуватися процеси програми. FADI запускає процеси програми на вказаних вузлах та періодично запускає їх контрольні точки, щоб зберегти образ виконання разом з будь-якими міжпроцесними повідомленнями у стабільному сховищі .

Вузли, що беруть участь у виконанні програми, постійно контролюються, і у разі збою вузла контрольні точки всіх процесів, що працюють на несправному вузлі, витягуються зі стабільного сховища, а процеси перезапускаються на робочому вузлі. Міжпроцесні повідомлення, отримані з моменту останньої контрольної точки, також відтворюються з журналу [23]. FADI також виявляє користувацькі процеси, які передчасно завершилися через тимчасовий збій обладнання, і аналогічним чином відновлює їх.

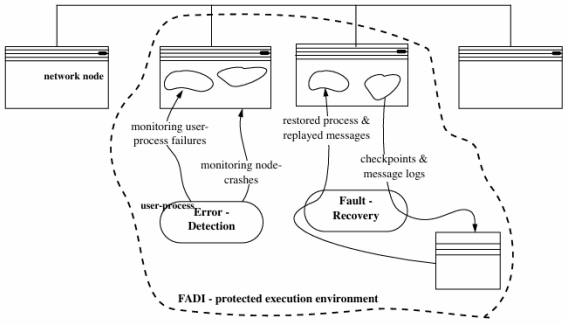

Рисунок 2.1 - Загальна конструкція FADI

Відправною точкою для всіх відмовостійких стратегій є виявлення помилкового стану, який за відсутності будь-яких коригувальних дій може призвести до збою системи. Механізм виявлення помилок FADI (EDM) визначає два типи апаратних збоїв: збої вузла процесора (викликані відключенням живлення) та тимчасові апаратні збої (тимчасові перемикання пам'яті, помилки шини тощо), які спричиняють збій окремого процесу програми, а також дозволяє інтеграцію користувацьких (специфічних для програми) перевірок на помилки.

Передбачається, що вузли обробки є безвідмовними, тобто вони

надсилають лише правильні повідомлення або взагалі нічого. Однак, процеси не обов'язково повинні бути безвідмовними, тобто з нульовою затримкою помилок [24].

Виявлення збоїв вузлів базується на завданні центрального моніторингу вузлів, яке періодично надсилає запити на підтвердження всім вузлам системи. Кожен вузол повинен підтвердити отримання протягом заздалегідь визначеного інтервалу часу (тайм-аут підтвердження), інакше він вважатиметься таким, що «вийшов з ладу».

Складність, пов'язана з цим методом виявлення, полягає в обчисленні часу очікування підтвердження. Дослідження в системі Delta-4 [4] стверджували, що загальний трафік повідомлень між компонентами динамічно розвивається системи (наприклад, спільного каналу зв'язку, такого як Ethernet) на практиці не можна вважати детерміністичним і він змінюється від екземпляра до екземпляра. Щоб врахувати цю варіацію, було реалізовано фонове завдання для динамічного обчислення часу обміну повідомленнями *(rtt)* між вузлами відносно поточного мережевого трафіку. Кожен новий зразок часу обміну повідомленнями *(Srtt)* фільтрується в « *згладжену* » оцінку за формулою:

$$\mathrm{RTT_{i+1}} = \alpha * \mathrm{RTT_i} + (1 - \alpha) * \mathrm{Srtt} \qquad (2.1)$$

де *RTT.* – поточна оцінка часу передачі даних туди й назад,

RTT. – нове обчислене значення,

а *a* – а

константа між 0 та 1, яка контролює, як швидко оцінений *rtt* адаптується до зміни навантаження мережі, і вибирається таким чином, щоб оцінка була більш чутливою до тенденцій зростання затримки та менш чутливою до тенденцій зниження. Розрахунок оптимального значення *a* виходить за рамки цієї роботи.

Тимчасові збої обладнання можуть спричинити збій процесів програми. Більшість цих помилок виявляються механізмами, вбудованими в апаратне забезпечення системи та ядро операційної системи [25]. Ці механізми включають

пастки для недійсних інструкцій, помилки шини, помилки сегментації, переривання, арифметичні винятки тощо.

Після виявлення помилок ядро ОС завершує роботу ураженого процесу, видаючи номер сигналу, що відповідає типу помилки. Коли користувацький процес завершується, FADI аналізує стан завершення процесу, щоб визначити, чи завершився він нормально, чи передчасно через збій, і в цьому випадку ініціюється відновлення процесу після збою.

Щодо виявлення помилок за допомогою користувача , для їх виявлення було виділено спеціальний обробник сигналів [5]. Все, що потрібно зробити програмісту, це викликати переривання із заздалегідь визначеним номером сигналу, і механізм виявлення обробить помилку так, ніби вона була викликана механізмом виявлення ядра (ОС) (KDM).

Для централізованого механізму виявлення, такого як FADI, важливо враховувати затримку виявлення помилок на вузлах розподіленої системи. Окрім часу, необхідного для доставки звіту про несправність до центрального сервера виявлення (приблизно половина часу обміну даними в мережі), нам також потрібно враховувати час реакції ядра ОС на тимчасові апаратні збої. Останній обчислюється один раз за допомогою завдання типу *черв'як* , яке імітує тимчасову несправність для вимірювання часу реакції ядра ОС. Час обміну даними обчислюється динамічно, як пояснено в попередньому розділі. Отже, ми можемо розрахувати затримку помилки виявлення збоїв процесу з 99% впевненістю за формулою:

$$t_{latency} = E(t_{edm}) + 3 * \sigma(t_{edm}) + \frac{rtt}{2} \qquad (2.2)$$

де:

E(Tedm) – середній час реакції механізму виявлення помилок ядра;

o(Tedm) – дисперсія Tedm;

rtt – поточний приблизний час передачі даних від головного вузла до активних вузлів.

Методи відмовостійкості для розподілених систем розвивалися у двох напрямках: відновлення з контрольними точками/відкатом та механізми реплікації процесів. Методи реплікації процесів широко вивчалися багатьма дослідниками [6] [7]. За допомогою таких методів кожен процес реплікується та виконується на окремому комп'ютері, так що ймовірність того, що всі репліки вийдуть з ладу, є прийнятно малою. Хоча ці методи призводять до меншого зниження продуктивності порівняно з механізмами контрольних точок, вони не є безвідмовними. Безвідмовні накладні витрати спричинені використанням механізмів багатоадресної розсилки для доставки кожного вихідного повідомлення до *групи* реплікованих процесів призначення [8]. Однак основною перешкодою для широкого впровадження методів реплікації процесів у різних областях є висока вартість надлишкового обладнання, необхідного для виконання реплік.

На відміну від механізмів реплікації процесів, цей метод не вимагає дублювання базового обладнання або реплікації процесів програми. Натомість кожен процес періодично записує свій поточний стан та/або деяку історію системи у стабільному сховищі, дія, яка називається *контрольною точкою.* У разі збою процеси повертаються до попередньої контрольної точки (*відкат)* і відновлюють своє виконання з цієї контрольної точки. Накладні витрати, які несе цей метод, більші, ніж у механізмів реплікації процесів, оскільки контрольні точки встановлюються під час безвідмовної роботи, а відкат вимагає певних дій для забезпечення послідовного відновлення, коли процеси аварійно завершують роботу. Тим не менш, ці накладні витрати зменшуються, оскільки комп'ютери та комунікаційні мережі стають ефективнішими. Отже, погіршення продуктивності, спричинене механізмом контрольної точки, є цілком прийнятним для класу програм, на які орієнтований FADI.

Ми обговоримо техніку резервного копіювання та відновлення FADI у два етапи: перший розглядає процес збереження стану виконання у стабільне сховище та його відновлення у разі збою. Другий досліджує координацію контрольних точок та відкату в розподіленому середовищі з урахуванням можливої міжпроцесної взаємодії.

Механізм контрольних точок FADI базується на техніці, розробленій *Condor ,* відомій як *контрольна точка Bytestream* [9]. Ідея полягає в тому, що процес програми з контрольною точкою повинен записувати інформацію про свій стан безпосередньо у файловий дескриптор, а процес, що перезапускається, повинен зчитувати цю інформацію безпосередньо з файлового дескриптора.

Для цільового класу довготривалих програм зменшення накладних витрат на безвідмовність має суттєве значення. Контрольні точки сприяють цим накладним витратам, призупиняючи виконання прикладної програми, поки контрольна точка визначається та записується на диск. У спробі мінімізувати накладні витрати на контрольні точки було введено техніку *неблокуючої* контрольної точки [26]. Створюється копія простору даних програми, і для виконання процедур контрольної точки використовується асинхронний потік керування, тобто зчитування стану процесу та запис його на диск, поки процес користувача продовжує виконання програмного коду.

На рисунку 2.2 наведено загальний огляд механізму контрольних точок FADI та відкату. Під час нормального запуску (запуску) користувацької програми, пролог контрольних точок FADI (пов'язаний з процесом з контрольними точками) встановлює локальний системний таймер для виклику контрольних точок користувацького процесу через регулярні проміжки часу, вказані користувачем у параметрах, а потім викликає процедуру user-main().

Встановлення контрольних точок починається із запису інформації про стан сигналу процесу (заблоковано, проігноровано, оброблено тощо) та статус відкритих файлів (режим, у якому відкрито, зміщення, за яким розташуванням) у структуру даних, виділену в купі процесу – отже, частину сегмента даних. Потім весь сегмент даних записується у файл контрольної точки в момент контрольної точки та зчитується назад у той самий адресний простір під час перезапуску у разі збою [27].

Для цього потрібна початкова та кінцева адреса сегмента даних. Початкова адреса - це константа , яка зазвичай залежить від платформи та може бути знайдена як директива компонувальника на сторінках довідки. Кінцева адреса фактично знаходиться на вершині купи та може бути повернута

системним викликом sbrk() у UNIX.

Збереження стеку вимагає збереження та відновлення *контексту стеку* (структури даних, що містить покажчик стеку серед іншої інформації, пов'язаної зі стеком), а також *фактичних даних* , які складають сам стек.

Для збереження та відновлення контексту стеку використовуються стандартні функції *"C"* setjmp() та longjmp(). setjmp() викликається для збереження поточного контексту стеку в буфер. Якщо потім викликається longjmp() з вказівником на збережений буфер, контекст стеку відновлюється, і виконання повертається в коді до точки, де була створена оригінальна setjmp(). Для збереження даних стеку потрібні початкова та кінцева точки стеку. Початок стеку - це добре відоме статичне місце, яке визначається як константа в специфікації ОС, кінець стеку, за визначенням, вказується вказівником стеку, що зберігається у збереженому контексті стеку.

Відновлення стеку є складнішим, оскільки простір стеку використовується (для локальних змінних) під час заміни. Безпосередня заміна простору стеку процесу простором, збереженим у файлі контрольної точки, призведе до незворотного пошкодження виконуваного образу. Щоб уникнути цього, вказівник стеку переміщується в безпечний буфер, зарезервований в області даних процесу. Переміщення вказівника стеку здійснюється за допомогою ще одного виклику setjmp(), вручну маніпулюючи вказівником стеку в збереженому контексті, щоб він вказував на наш буфер в області даних, а потім longjmp(). Потім зарезервований простір в області даних використовується як тимчасовий стек для безпечної заміни початкової області стеку процесу на ту, що була раніше збережена у файлі контрольної точки. Таким чином, під час переміщення точки стеку виконання назад у початкове місце, ми фактично використовуватимемо відновлений стек.

FADI підтримує цілісність користувацьких файлів протягом усього процесу контрольних точок/відкатів, відстежуючи додавання та оновлення користувацьких файлів. Це досягається шляхом доповнення системних викликів UNIX та функцій бібліотеки *"C"* , які записують у користувацькі файли (наприклад, write, fwrite, printf тощо), власними функціями, які додають файл до

списку перед викликом оригінальної процедури. Під час контрольної точки файли в цьому списку перевіряються, і якщо файл був відкритий для оновлення, то створюється тіньова копія файлу, інакше файл був відкритий для додавання, і робиться позначка про його поточний розмір. Коли контрольна точка зайнята, розгалужений потік завершується сигналом користувача.

Якщо виявлено апаратну помилку та запитується відкат користувацької програми, то пролог контрольної точки викликає процедуру відновлення. Спочатку вона замінює користувацькі файли, відкриті для оновлення, їхніми тіньовими копіями та обрізає файли, відкриті для додавання до розміру, записаного на останній контрольній точці. Ці операції виконуються лише в тому випадку, якщо вміст файлу було змінено з моменту останньої контрольної точки [28]. Потім сегмент даних замінюється тим, що зберігається у файлі контрольної точки. Тепер процедура відновлення має список відкритих файлів, обробників сигналів тощо у власному просторі даних та відновлює ці частини стану програми. Далі вона перемикає поточний стек програми на той, що був збережений на момент контрольної точки, та повертається до користувацького коду на тій самій інструкції, яка була перервана під час виклику останньої контрольної точки. Нарешті, оскільки контрольна точка була виконана обробником сигналів, ОС UNIX відновлює всі регістри процесора до їх стану до контрольної точки.

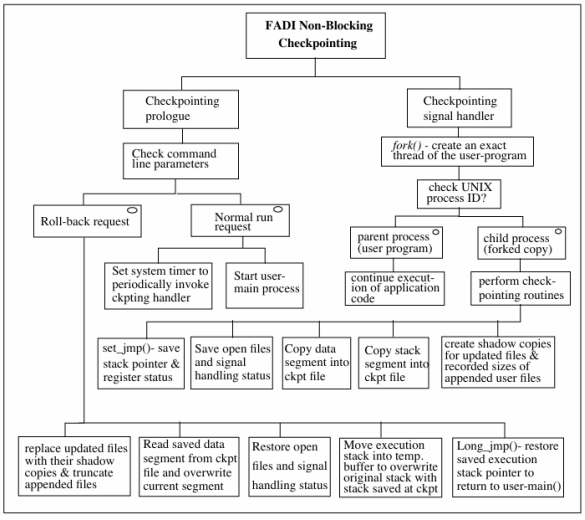

Рисунок 2.2 - Створення контрольних точок та відкат у FADI

2.2 Методи виявлення, локалізації та відновлення після збоїв

Метою системи FRIENDS було забезпечення механізмів для створення відмовостійких застосунків більш гнучким способом. Гнучкість досягається завдяки наданню об'єктно-орієнтованих бібліотек метаоб'єктів , а також завдяки наданню підсистем на платформі мікроядра. Підсистеми можна повторно використовувати як є, а також портувати на різні платформи. Бібліотеки метаоб'єктів можна швидко портувати на нову платформу (доступність компілятора) та розширювати за допомогою методів об'єктно-орієнтованої розробки. Метарівневий підхід, що використовується тут, надає нові засоби для розробки функціональності, яка традиційно присутня в операційній системі, як метарівневе програмне забезпечення.

Архітектура системи (див. рис. 2.3) складається з кількох шарів: (і) шар ядра, який може бути або ядром Unix, або, краще, мікроядром, таким як Chorus, (ii) системний шар, що складається з кількох спеціалізованих підсистем, по одній для кожної абстракції, і, нарешті, (iii) шар користувача, призначений для реалізації застосунків.

Системний рівень організований як набір підсистем. У технології мікроядра підсистема відповідає набору сервісів, що реалізують будь-яку програмну систему, наприклад, операційну систему поверх мікроядра (наприклад, Unix на Chorus). У FRIENDS кожна підсистема надає сервіси для забезпечення відмовостійкості, безпечного зв'язку або розповсюдження. Будь-яка підсистема може бути реалізована апаратно та програмно . Три необхідні підсистеми є наступними:

- *FTS* (підсистема відмовостійкості) надає основні послуги, обов'язкові для відмовостійких обчислень, зокрема виявлення помилок та підозрювання на збої , які мають бути реалізовані як низькорівневі сутності. Ця підсистема також включає засоби керування доменами конфігурації та реплікації, а також стабільну підтримку сховища.
- *SCS* (Підсистема безпечного зв'язку) надає основні послуги, які, очевидно, мають бути реалізовані як довірені об'єкти в системі (поняття

довіреної обчислювальної бази). Ці послуги повинні включати , зокрема, сервер автентифікації, а також сервер авторизації , сервер каталогів та сервер аудиту.

- *GDS* (підсистема розподілу на основі груп) надає базові послуги, реалізуючи підтримку розподілу для об'єктно-орієнтованих програм, де об'єкти можна реплікувати.

Ці основні послуги включають засоби управління групами та протоколи атомарної багатоадресної розсилки.

Підсистеми надають основні послуги, необхідні механізмам, реалізованим за допомогою метаоб'єктів. Ці послуги можна розглядати як *програмно-замінні блоки* (SRU). Використовуючи технологію мікроядра, системний рівень може бути легко складений з необхідних підсистем, кожна з яких використовує відповідні SRU [29].

Користувацький рівень поділяється на два підрівні: прикладний рівень та метаоб'єктний рівень, що контролює поведінку прикладних об'єктів. Деякі бібліотеки класів метаоб'єктів для реалізації відмовостійких та безпечних прикладних програм реалізуються поверх відповідної підсистеми та надають користувачеві механізми, які можна налаштовувати за допомогою об'єктно-орієнтованих методів.

- *libft mo* надає класи метаоб'єктів для різних стратегій відмовостійкості (на основі стабільного зберігання або реплікації) з урахуванням фізичних несправностей, враховуючи вузли, що не працюють без збоїв.
- *бібліотека mo* надає класи метаоб'єктів для різних протоколів безпечного зв'язку, використовуючи методи шифрування, обчислення та перевірку підписів на основі криптосистем із секретним або відкритим ключем.
- *бібліотека mo* надає класи метаоб'єктів для обробки взаємодії віддалених об'єктів, яку можна реалізувати за допомогою груп. Поєднання цих метаоб'єктів та GDS забезпечує підтримку розподілених об'єктно-орієнтованих програм під час виконання.

Для стратегій активної реплікації ми припускаємо, що об'єкти застосунку мають «детерміністичну поведінку». Детермінована поведінка гарантує, що всі одержувачі, які обробляють однакові вхідні повідомлення, отримують однакові

результати. Оскільки атомарна багатоадресна розсилка гарантує, що всі правильні одержувачі отримують однакові вхідні повідомлення в одному порядку, будь-який виклик методу призведе до однакових результатів для будь-якої з реплік об'єктів. Паралелізм та інші джерела недетермінізму ще не розглядалися.

Статичний вигляд загальної архітектури системи проілюстровано на рисунку 2.3. FRIENDS надає набір підсистем та кілька бібліотек класів метаоб'єктів .

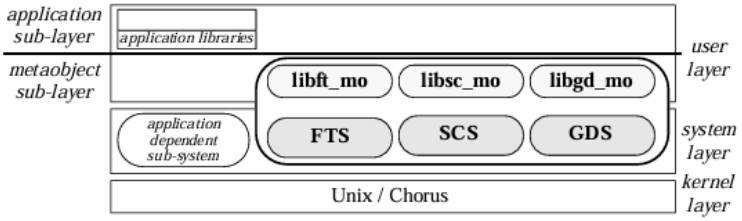

Рисунок 2.3 - Загальна архітектура системи

Реалізація будь-якої абстракції (відмовостійкість, безпечний зв'язок, розподіл), таким чином, поділяється на бібліотеку класів метаоб'єктів та відповідну підсистему, таким чином охоплюючи принаймні частково користувацький та системний рівні.

Для кожного об'єкта в застосунку FRIENDS можна використовувати один або кілька метаоб'єктів. Коли потрібен лише розподіл, на метарівні використовуються лише метаоб'єкти зв'язку [30]. Коли потрібна розподілена відмовостійкість, то на рівні застосунку оголошуються метаоб'єкти відмовостійкості відповідно до бажаної стратегії. Розподіл тепер обробляється метаоб'єктами відмовостійкості з використанням метаметаоб'єктів. З точки зору програміста метаоб'єктів відмовостійкості, розподіл обробляється на метарівні за допомогою метаоб'єктів зв'язку. Коли необхідна безпека, то замість використання стандартних метаоб'єктів зв'язку, програміст відмовостійкості оголошує метаоб'єкти безпеки. Ці метаоб'єкти забезпечують безпеку на рівні зв'язку та використовують переваги метаоб'єктного підходу для віддаленого зв'язку. Таке рекурсивне використання метаоб'єктів призводить до кількох

метарівнів у кінцевому застосунку. У наступних розділах ми опишемо використання метаоб'єктів для обробки розподілу, безпеки зв'язку та відмовостійкості, включаючи протоколи міжреплікації.

Додаток розглядається як сукупність взаємодіючих об'єктів, розроблених за допомогою об'єктно-орієнтованої мови програмування (наразі C++). Кожен об'єкт додатку відображається GDS на об'єкт середовища виконання, залежно від сутностей, що обробляються базовою операційною системою (процеси Unix або актори Chorus). Кожен об'єкт середовища виконання не лише складається з об'єкта додатку, але й містить один або кілька метаоб'єктів в одному адресному просторі.

Лістинг 2.1 - Набір метаоб'єктів

```
Runtime_object = {A, FT, SC, GD} with:
     A  : application object
     FT : fault tolerance metaobject
     SC : secure communication metaobject
     GD : group-based distribution metaobject
```

Набір метаоб'єктів залежить від властивостей, які необхідно надати застосунку, та включає принаймні один метаоб'єкт, GD, для розподіленої взаємодії [31]. В ідеалі, додавання властивостей до застосунку передбачає додавання інших метаоб'єктів до набору. Поняття *набору метаоб'єктів* подібне до поняття *метапростору ,* визначеного в Apertos, а також до поняття *відбивної вежі об'єктів* в ABCL/. На рисунку 2.4 нижче зображено нашу модель застосунку на конфігурації розподіленої системи.

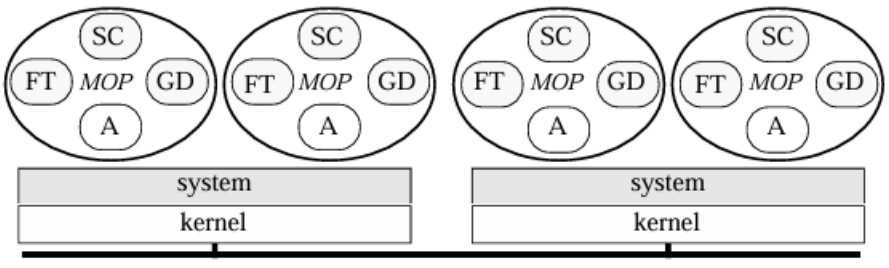

Рисунок 2.4 - Розподілений застосунок з використанням FRIENDS

В межах одного об'єкта середовища виконання взаємодія між об'єктом програми та метаоб'єктами здійснюється через МОР. Взаємодія об'єктів середовища виконання базується на моделі клієнт-сервер. Точніше, вона базується на моделі проксі: об'єкт сервера сприймається як проксі в адресному просторі клієнта. До проксі-сервера підключено метаоб'єкт , що обробляє клієнтську сторону, до ефективного сервера підключено метаоб'єкт, що обробляє серверну сторону. Будь-яка абстракція обробляється протоколом між двома метаоб'єктами. Програмісти програми просто пишуть об'єкти програми та вибирають відповідний набір метаоб'єктів.

Для вирішення проблем, було визначено *трирівневу модель застосунку. Будь-який (виконуваний) об'єкт* організований з використанням кількох рівнів: перший рівень або базовий рівень (об'єкт застосунку), кілька проміжних метарівнів (метаоб'єкти для відмовостійкості та безпечного зв'язку) і, нарешті, останній метарівень, відповідальний за обробку взаємодії об'єктів. Така структура застосунку передбачає послідовність взаємодій через МОР, як показано на рисунку 2.5. Набір метаоб'єктів організовано у вигляді *стеку.*

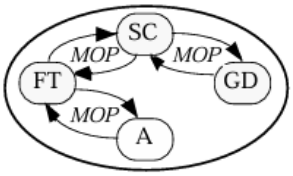

Рисунок 2.5 - Взаємодія базового та метарівня

Мінімальні специфікації МОР, які нам потрібні та які ми використовували, є наступними:

- створення/видалення об'єкта: створення об'єкта передбачає створення кількох реплік, реєстрацію в комунікаційній групі та автентифікацію за потреби;
- перехоплення викликів: семантика виклику методу може бути реалізована по-різному, наприклад, відповідно до заданої стратегії відмовостійкості ;

• доступ базового рівня: методи та атрибути базового рівня можна маніпулювати з метарівня відповідно для виконання методів об'єкта на активних репліках та отримання стану об'єкта.

Розглядаючи лише один розподіл обробки на метарівні, будь-який виклик методу сервера перехоплюється метаоб'єктом клієнта на стороні клієнта, повідомлення про виклик пересилається на сторону сервера, це повідомлення отримує метаоб'єкт сервера, і, нарешті, метод виконується на базовому рівні. Цей протокол проілюстровано на рисунку 2.6 нижче. Приклад, реалізований з використанням цієї моделі.

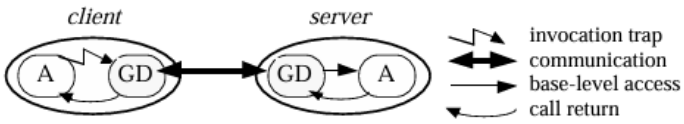

Рисунок 2.6 - Використання метаоб'єктів для розповсюдження

Враховуючи тепер кілька проміжних рівнів, кожен виклик методу сервера перехоплюється наступним метарівнем. Це рекурсивно виконується до останнього метарівня (GD), де повідомлення про виклик пересилається на сервер. Повідомлення про виклик отримується метарівнем сервера (GD) , і виклик рекурсивно поширюється через проміжні рівні до базового рівня, де виконується метод. Це проілюстровано на рисунку 2.7 нижче.

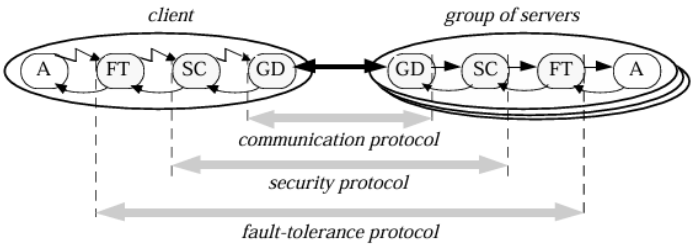

Рисунок 2.7 - Багаторівнева реалізація клієнт-серверного протоколу

На цьому рисунку також показано різні базові протоколи між клієнтським об'єктом та реплікованим серверним об'єктом [32]. Завдяки цій структурі

застосунку будь-який проміжний метарівень можна досить легко додавати або видаляти, таким чином додаючи або видаляючи відповідний базовий протокол.

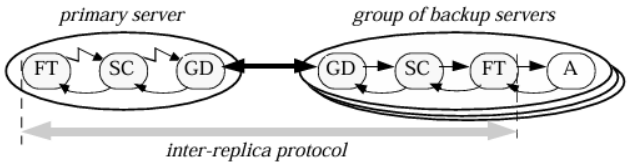

Рисунок 2.8 - Багаторівнева реалізація протоколу первинного резервного копіювання

Ця багаторівнева модель також використовується в реалізації протоколів міжреплікації на стороні сервера. Наприклад, взаємодія між основним об'єктом та групою резервних реплік така, що основний об'єкт є клієнтом групи резервних реплік. Основна репліка фіксує стан об'єкта базового рівня та прозоро викликає update_state . метод резервних копій (рисунок 2.8). Це можливо, оскільки в багатьох стратегіях реплікації протокол між репліками є протоколом клієнт-сервер.

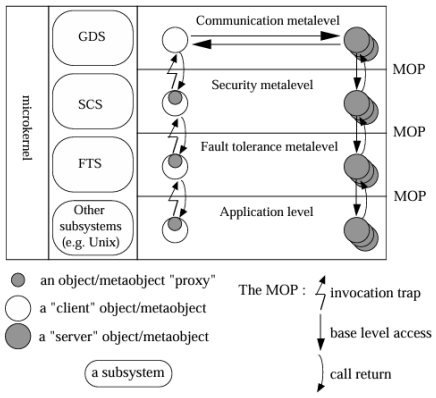

Рисунок 2.9 - Системний підхід FRIENDS

Загальна архітектура та модель застосунку з використанням метаоб'єктів зображена на рисунку 2.9. Цей рисунок ілюструє систему FRIENDS з іншої точки зору. Структурування застосунку виділено та показує, що та сама модель використовується для програмування на будь-якому рівні [33]. Об'єкти та метаоб'єкти, а також метаоб'єкти між собою взаємодіють через MOP. Цей

рисунок також показує, що метаоб'єкти покладаються на базові сервіси, що працюють поверх мікроядра.

2.3 Технології реплікації, балансування навантаження та забезпечення доступності

У розподілених системах реального часу відмовостійкість має вирішальне значення для забезпечення того, щоб система продовжувала працювати коректно навіть за наявності збоїв, дотримуючись суворих часових обмежень. Механізми відмовостійкості в цих системах повинні гарантувати доступність системи, правильну роботу та здатність відновлюватися після збоїв без порушення термінів реального часу. У цьому розділі детально розглядаються основні механізми відмовостійкості, що використовуються в розподілених системах реального часу, включаючи резервування, алгоритми консенсусу, стратегії виявлення та відновлення помилок, а також методи відмовостійкого планування.

Надмірність та реплікація є одними з найпоширеніших механізмів досягнення відмовостійкості в розподілених системах. Ці методи передбачають дублювання критичних компонентів системи, таких як дані, служби або обчислювальні процеси, так що у разі відмови одного компонента інший може взяти на себе його роботу, забезпечуючи працездатність системи.

Види резервування:

Реплікація даних:

У розподілених системах дані реплікуються на кілька серверів або вузлів, щоб забезпечити високу доступність та відмовостійкість. Це особливо важливо для систем, де цілісність та доступність даних є надзвичайно важливими.

Реплікація первинного та резервного вузлів: один вузол призначається як первинний (головний), а інші діють як резервні. Якщо первинний вузол виходить з ладу, один із резервних вузлів перетворюється на первинний.

Багатомастерна реплікація: усі вузли в системі мають можливість обробляти запити та зберігати копії даних. Такий підхід підвищує доступність та

розподіл навантаження, але ускладнює підтримку узгодженості.

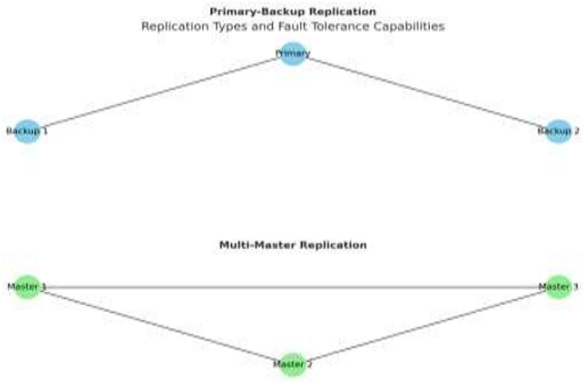

Рисунок 2.10 - Це схема, що ілюструє два типи реплікації

Реплікація служби:

Активна реплікація: Кілька копій служби або процесу працюють одночасно, кожна з яких обробляє вхідні запити. Якщо одна репліка виходить з ладу, інші продовжують надавати послугу без перебоїв. Резервна реплікація: Подібна до резервних серверів, але резервні репліки неактивні, доки вони не знадобляться. Ця модель зменшує використання ресурсів, але може призвести до затримки відновлення після збоїв.

Проблеми з резервуванням:

Узгодженість проти доступності: У реплікованих системах забезпечення узгодженості між репліками (тобто ідентичності всіх копій даних) може бути складним завданням, особливо за наявності мережевих розділів або одночасних оновлень.

Затримка: Реплікація може призвести до затримки, особливо під час синхронізації даних між репліками в системах реального часу. Це критична проблема в системах, де потрібні відповіді з низькою затримкою.

Реплікація основної резервної копії:

Основний вузол синхронізує дані з резервними вузлами.

Відмовостійкість досягається шляхом перемикання на резервний вузол у разі виходу з ладу основного. Реплікація з кількома головними вузлами:

Кілька головних вузлів синхронізуються один з одним.

Відмовостійкість підвищується, оскільки будь-який головний пристрій може обробляти запити у разі збою іншого [34]. Стрілки позначають шляхи синхронізації даних.

Алгоритми консенсусу є фундаментальними для підтримки узгодженості в розподіленій системі, особливо в умовах збоїв. Ці алгоритми дозволяють розподіленим вузлам узгоджувати одне значення або стан, гарантуючи, що система залишається узгодженою, навіть якщо деякі вузли виходять з ладу або поводяться неправильно.

Протокол Паксос:

Paxos - це алгоритм консенсусу, розроблений для досягнення згоди між групою вузлів, навіть за наявності збоїв. Він гарантує, що розподілена система може досягти консенсусу щодо одного значення, навіть якщо деякі вузли виходять з ладу або відбувається розщеплення мережі.

Слабкі сторони: Paxos є відносно складним і може створювати значні накладні витрати на повідомлення, особливо у великих системах.

Алгоритм плоту:

Raft - це альтернатива Paxos, яку легше зрозуміти та впровадити, водночас забезпечуючи ті ж гарантії консенсусу. Вона широко використовується в системах реального часу, таких як Kubernetes та Etcd.

Raft організовує вузли за моделлю «лідер-послідовник», де лідируючий вузол відповідає за підтримку стану системи та координацію змін.

Реплікація з позначкою перегляду:

Цей алгоритм покращує Paxos, зменшуючи потребу в кругових циклах зв'язку, покращуючи продуктивність та відмовостійкість для великомасштабних систем.

Таблиця 2.1

Порівняння для Paxos, Raft та Viewstamped Replication у розподілених системах реального часу:

Алгоритм	Сильні сторони	Слабкі сторони	Ідеальний сценарій використання
Paxos	- Строга узгодженість. - Добре вивчений і перевірений.	- Складна реалізація. - Високі накладні витрати на зв'язок.	- Системи, що потребують строгої узгодженості (наприклад, розподілені БД).
Raft	- Легше зрозуміти та впровадити. - Ефективний вибір лідера.	- Менша відмовостійкість в екстремальних умовах. - Все ще потребує значних витрат на зв'язок.	- Розподілені системи реального часу, сховища ключ-значення, мікросервіси.
Viewstamped Replication	- Висока доступність. - Швидка зміна лідера.	- Потребує більше комунікацій для вибору лідера. - Може страждати від поділу мережі (partitions).	- Додатки реального часу з потребою в низькій затримці (репліковані БД, хмарні сервіси).
Paxos	- Строга узгодженість. - Добре вивчений і перевірений.	- Складна реалізація. - Високі накладні витрати на зв'язок.	- Системи, що потребують строгої узгодженості (наприклад, розподілені БД).
Raft	- Легше зрозуміти та впровадити. - Ефективний вибір лідера.	- Менша відмовостійкість в екстремальних умовах. - Все ще потребує значних витрат на зв'язок.	- Розподілені системи реального часу, сховища ключ-значення, мікросервіси.
Viewstamped Replication	- Висока доступність. - Швидка зміна лідера.	- Потребує більше комунікацій для вибору лідера. - Може страждати від поділу мережі (partitions).	- Додатки реального часу з потребою в низькій затримці (репліковані БД, хмарні сервіси).
Paxos	- Строга узгодженість. - Добре вивчений і перевірений.	- Складна реалізація. - Високі накладні витрати на зв'язок.	- Системи, що потребують строгої узгодженості (наприклад, розподілені БД).

Виклики:

Вибір лідера: У консенсусних алгоритмах, таких як Raft, збій у вузлі

лідера вимагає нових виборів лідера, що може призвести до затримки. У системах реального часу мінімізація цієї затримки є вирішальною для дотримання термінів.

Масштабованість: Зі збільшенням кількості вузлів у розподіленій системі збільшуються накладні витрати на досягнення консенсусу, що впливає на продуктивність системи та гарантії роботи в режимі реального часу.

Системи реального часу повинні включати надійні механізми виявлення помилок, щоб ідентифікувати несправності одразу після їх виникнення, забезпечуючи своєчасне відновлення та запобігаючи збоям системи. Для виявлення помилок та відновлення після них використовується кілька стратегій:

Механізми серцебиття:

Механізми серцевого ритму періодично надсилають сигнали між компонентами системи для підтвердження їхнього робочого стану. Якщо компонент пропускає серцевий ритм, вважається, що він вийшов з ладу, і вживаються коригувальні дії.

Приклад: У розподілених базах даних, якщо вузол не надсилає сигнал серцевого ритму, система може позначити його як недоступний та перенаправити запити до справних реплік.

Сторожові таймери:

Сторожовий таймер контролює продуктивність системи та запускає процедури відновлення, коли перевищено певні порогові значення (наприклад, пропущені терміни або невідповідність).

Приклад: У критично важливих для безпеки системах реального часу, таких як медичні пристрої, сторожові таймери використовуються для моніторингу критичних процесів і скидання системи, якщо вона не працює в межах прийнятних параметрів.

Відкат та контрольні точки:

Контрольні точки включають періодичне збереження стану системи, що дозволяє системі повернутися до відомого справного стану у разі збою.

Відновлення після відкату особливо корисне для забезпечення можливості відновлення критично важливих операцій без втрати даних або

невідповідностей.

Виклики:

Затримка відновлення: Механізми відновлення, такі як відкат та контрольні точки, можуть спричиняти затримки, які можуть перешкоджати дотриманню термінів у реальному часі. Забезпечення швидкого відновлення без порушення термінів є суттєвою проблемою в системах реального часу.

Накладні витрати: Механізми постійного виявлення та відновлення помилок створюють обчислювальні та комунікаційні накладні витрати, що може знизити продуктивність системи та збільшити споживання ресурсів.

У системах реального часу алгоритми планування використовуються для розподілу ресурсів та забезпечення дотримання термінів виконання завдань. У разі виникнення збоїв система повинна адаптувати своє планування, щоб врахувати перерозподіл та відновлення ресурсів, не ставлячи під загрозу обмеження реального часу.

Монотонне планування за швидкістю (RMS):

Це алгоритм планування з фіксованим пріоритетом, де завданням з коротшими періодами (вищою частотою) надається вищий пріоритет. RMS широко використовується в операційних системах реального часу завдяки своїй простоті та передбачуваності.

Відмовостійкі модифікації: У відмовостійких системах модифікації RMS можуть включати перепризначення завдань у разі відмови вузла, що гарантує, що завдання продовжуватимуть дотримуватися термінів. Найшвидший термін виконання (EDF):

EDF - це алгоритм динамічного планування з пріоритетами, де завданням надається пріоритет на основі їхніх термінів виконання, причому першим планується найдавніший термін виконання. Цей підхід є оптимальним для однопроцесорних систем, але може бути складнішим у розподіленому середовищі.

Відмовостійкі модифікації: У розподілених системах відмовостійкі модифікації можуть включати перепризначення завдань різним процесорам або динамічне коригування планування залежно від стану системи.

Відмовостійкі алгоритми планування:

Для обробки міграції, реплікації та відновлення завдань у разі виникнення збоїв було розроблено спеціалізовані алгоритми [35]. Ці алгоритми гарантують, що завдання продовжуватимуть виконуватися з використанням доступних ресурсів, навіть якщо деякі компоненти вийдуть з ладу.

Приклад: В аерокосмічній системі, якщо вузол обробки виходить з ладу, відмовостійкий планувальник динамічно перепризначає завдання резервному вузлу, гарантуючи, що не буде пропущено жодних критично важливих термінів. Проблеми:

Динамічна адаптація: Планування в режимі реального часу має бути достатньо гнучким, щоб динамічно адаптуватися до збоїв, дотримуючись при цьому термінів. Балансування вимог реального часу з необхідністю відновлення після помилок є постійним викликом.

Розподіл ресурсів: Ефективне управління обмеженими ресурсами під час відновлення після збоїв має вирішальне значення для відмовостійкого планування. Надмірне виділення ресурсів для відновлення може знижувати продуктивність системи та порушувати роботу в режимі реального часу.

Ці механізми - резервування, консенсусні алгоритми, виявлення помилок, стратегії відновлення та відмовостійке планування - формують основу відмовостійкості в розподілених системах реального часу. Однак їхня ефективність залежить від того, наскільки добре вони інтегровані в систему та як вони балансують потребу в надійності з вимогами системи до продуктивності в реальному часі.

За допомогою таких візуальних посібників, як графіки, таблиці та схеми, цей розділ забезпечує повне розуміння механізмів відмовостійкості, які допомагають забезпечити високу доступність та низьку затримку реагування в критично важливих системах реального часу. Проблеми та компроміси, пов'язані з кожним механізмом, підкреслюють складність підтримки відмовостійкості без шкоди для основних цілей системи.

Розгортання відмовостійких механізмів у розподілених системах реального часу охоплює широкий спектр критично важливих застосувань, від

аерокосмічних систем та автономних транспортних засобів до промислового управління та охорони здоров'я. Ці системи повинні відповідати суворим обмеженням реального часу, забезпечуючи при цьому їхню можливість продовжувати коректне функціонування навіть за умови збоїв обладнання, програмних помилок або проблем із мережею. У цьому розділі ми розглянемо кілька тематичних досліджень та реальних застосувань, які показують, як відмовостійкі механізми реалізуються в розподілених системах та їхній вплив на продуктивність та надійність системи.

Аерокосмічні системи, такі як системи керування супутниками, космічні апарати та авіоніка літаків, є яскравими прикладами критично важливих систем реального часу, де відмовостійкість має першорядне значення. Ці системи розроблені для роботи в екстремальних умовах, де збої можуть призвести до катастрофічних наслідків [36]. Для підтримки цілісності роботи за будь-яких умов, механізми відмовостійкості повинні бути ретельно інтегровані як в апаратну, так і в програмну архітектури.

Тематичне дослідження: Марсохід (NASA)

Марсохід NASA, який працює на поверхні Марса, служить ідеальним прикладом відмовостійкої розподіленої системи реального часу. Марсохід оснащений резервними системами, включаючи резервні блоки зв'язку, живлення та обробки даних, для забезпечення безперервної функціональності у разі збоїв компонентів. Марсохід також використовує відмовостійкі алгоритми реального часу для адаптації своєї поведінки та коригування планування завдань у разі виникнення збою.

Резервування: Марсохід має кілька резервних блоків для ключових компонентів, таких як система зв'язку, акумулятор та процесори.

Планування в режимі реального часу: завдання плануються на основі пріоритетів та обмежень у реальному часі, що забезпечує своєчасне виконання критично важливих дій (наприклад, зв'язок із Землею або збір наукових даних).

Відновлення після збою: У разі виходу з ладу марсохід може переналаштувати свої системи або автономно переключитися на резервні компоненти.

Виклики:

Обмеження ресурсів: Обмежена обчислювальна потужність та енергія, доступні на марсоході, вимагають ретельного управління механізмами відмовостійкості.

Затримка: Через тривалу затримку зв'язку між Марсом і Землею, система повинна мати можливість працювати автономно, не покладаючись на інструкції в режимі реального часу від наземних контролерів.

Командний центр: Надсилає завдання марсоходу.

Планувальник завдань: розподіляє завдання між компонентами системи.

Основна система: Виконує основні функції, надаючи дані до масивів датчиків, процесорів даних та елементів керування виконавчими механізмами.

Резервна система: Забезпечує резервування для основної системи.

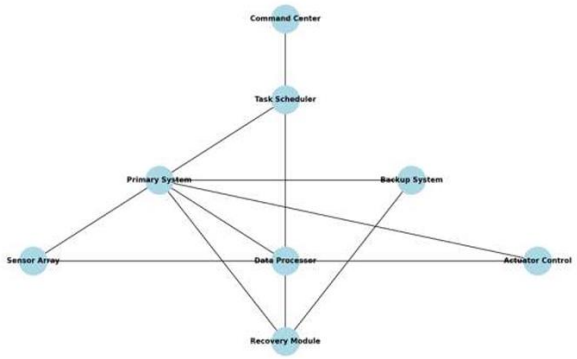

Рисунок 2.11 - Блок-схема, що відображає механізми резервування та відновлення після збоїв у системі марсохода

Масив датчиків, процесор даних та керування виконавчим механізмом: Виконання зондування навколишнього середовища, аналізу даних та механічних операцій.

Модуль відновлення: керує відновленням після збоїв та повторно інтегрує систему в роботу.

Стрілки ілюструють потік завдань, синхронізацію та шляхи відновлення.

Автономні транспортні засоби, включаючи безпілотні автомобілі та дрони, значною мірою залежать від розподілених систем у режимі реального часу, щоб приймати рішення за частки секунди на основі даних датчиків та умов

навколишнього середовища. Ці системи повинні бути здатні працювати безпечно та ефективно, навіть за наявності несправностей компонентів або погіршення даних датчиків, забезпечуючи при цьому продуктивність у режимі реального часу для відповідності вимогам безпеки.

Тематичне дослідження: Автономні автомобілі Waymo (Google)

Waymo, проект Google з розробки автономних транспортних засобів, інтегрує відмовостійкість на кількох рівнях своєї системної архітектури, щоб забезпечити безпеку та надійність в режимі реального часу. Транспортні засоби Waymo використовують комбінацію надлишкових датчиків, алгоритмів планування в режимі реального часу та відмовостійких протоколів зв'язку, щоб гарантувати безпечну роботу автомобіля навіть у разі виходу з ладу певних датчиків або систем.

Резервування: критично важливі датчики, такі як LiDAR, камери та GPS, дублюються, щоб забезпечити транспортний засіб достатньою кількістю інформації для прийняття рішень у разі виходу з ладу одного з датчиків.

Прийняття рішень у режимі реального часу: Система керування транспортним засобом використовує планування в режимі реального часу для визначення пріоритетів критичних дій, таких як уникнення перешкод або екстрене гальмування.

Виявлення та відновлення помилок: Транспортний засіб використовує методи виявлення помилок, такі як сторожові таймери, щоб забезпечити правильну роботу систем [37]. Якщо виявлено несправність, система може або спробувати відновитися, або безпечно зупинитися, щоб уникнути аварій.

Виклики:

Відмова датчика: Несправності датчиків або неточності даних можуть поставити під загрозу безпеку. Система повинна мати можливість виявляти та компенсувати відмови датчиків у режимі реального часу.

Затримка системи: автономні транспортні засоби повинні реагувати на події з мінімальною затримкою, щоб уникнути аварій. Механізми стійкості до відмов повинні бути розроблені таким чином, щоб мінімізувати затримку, зберігаючи при цьому безпеку системи. показує стратегії стійкості до відмов

автономних транспортних засобів, таких як Waymo:

Таблиця 2.2

Показує стратегії стійкості до відмов автономних транспортних засобів, таких як Waymo:

Система	Надмірність сенсорів	Відновлення після збоїв	Прийняття рішень у реальному часі
Waymo	LiDAR, камери, радар.	Надлишкові сенсори, об'єднання даних у реальному часі, втручання вручну.	Швидка адаптація до перешкод, висока ефективність.
Tesla Autopilot	Камери, радар, ультразвукові сенсори.	Об'єднання даних (sensor fusion), перехід на ручне керування.	Швидка реакція, але можливі невеликі затримки в складних ситуаціях.
Cruise	LiDAR, радар, камери, ультразвукові сенсори.	Надлишкові дані, механізми безпеки на випадок збою.	Швидке коригування в реальному часі, надійність у трафіку.
Aptiv	LiDAR, камери, радар, ультразвукові сенсори.	Вбудована надмірність, екстрена зупинка при критичному збої.	Прийняття рішень з високим рівнем впевненості, робота в різних дорожніх умовах.

Промислові системи керування, включаючи системи SCADA (диспетчерського керування та збору даних), управління енергосистемою та робототехніку у виробництві, є критично важливими розподіленими системами реального часу, що використовуються для моніторингу та контролю процесів у різних галузях промисловості. Ці системи вимагають безперервної роботи та повинні бути дуже стійкими до збоїв, оскільки будь-який простій може призвести до значних економічних втрат або загроз безпеці.

Тематичне дослідження: Системи управління енергомережею

В управлінні енергомережею відмовостійкість має вирішальне значення для підтримки стабільного та надійного електропостачання, особливо у разі відмов обладнання або стихійних лих. Сучасні енергомережі містять розподілені системи керування, які спираються на відмовостійкі механізми, щоб забезпечити ефективну роботу мережі навіть за наявності збоїв.

Резервування: Електромережі часто мають кілька вузлів виробництва та

розподілу електроенергії з резервними системами, що беруть на себе роботу у разі збоїв.

Алгоритми консенсусу: Розподілені протоколи прийняття рішень, такі як алгоритми консенсусу, гарантують, що всі частини мережі досягають згоди щодо стану системи, навіть якщо деякі вузли зазнають збоїв.

Моніторинг та відновлення в режимі реального часу: Механізми виявлення та відновлення несправностей використовуються для виявлення несправностей у режимі реального часу, таких як несправності обладнання, та ініціювання процедур відновлення для відновлення нормальної роботи мережі.

Виклики:

Масштабованість: електричні мережі можуть охоплювати величезні географічні області, що вимагає масштабованих рішень для забезпечення відмовостійкості, які можуть обробляти збої в різних регіонах мережі.

Синхронізація: Забезпечення синхронізації мережі під час відновлення після збоїв є ключовим завданням для підтримки безперервної роботи.

Системи охорони здоров'я, особливо ті, що керують відділеннями інтенсивної терапії, телемедициною та моніторингом медичного обладнання, стають дедалі більш залежними від розподілених систем у режимі реального часу. Відмовостійкість в охороні здоров'я є важливою для забезпечення постійного моніторингу даних пацієнтів та безперебійного проведення критично важливих процедур.

Тематичне дослідження: Системи моніторингу пацієнтів у режимі реального часу

У лікарнях системи моніторингу пацієнтів у режимі реального часу відстежують життєво важливі показники, такі як частота серцевих скорочень, артеріальний тиск і рівень кисню. Ці системи відповідають за те, щоб лікарі були в режимі реального часу попереджені про будь-які зміни у стані пацієнта, що загрожують життю [38]. Відмовостійкість цих систем має вирішальне значення для забезпечення безперервного моніторингу та своєчасного втручання.

Резервування: Критично важливі компоненти системи моніторингу, такі як датчики та сервери, реплікуються для забезпечення безперебійного збору та

аналізу даних.

Сповіщення в режимі реального часу: Система використовує методи планування в режимі реального часу та виявлення помилок для своєчасного надсилання сповіщень медичним працівникам у разі виявлення аномальних показників.

Відновлення після збою: Якщо датчик або мережеве з’єднання виходять з ладу, резервні системи автоматично вмикаються для підтримки безперервного моніторингу.

Виклики:

Цілісність даних: забезпечення точності та узгодженості даних пацієнтів за наявності збоїв мережі або обладнання.

Затримка: Затримки в передачі даних або генерації сповіщень можуть призвести до критичних затримок у догляді за пацієнтами.

Ці тематичні дослідження підкреслюють різноманітне застосування відмовостійких механізмів у розподілених системах реального часу. Чи то автономні транспортні засоби, енергетичні мережі чи системи охорони здоров'я, основні принципи резервування, консенсусу, виявлення помилок та відновлення після збоїв є критично важливими для забезпечення надійності системи та дотримання суворих обмежень реального часу. Однак проблеми, пов'язані з управлінням ресурсами, затримкою та масштабованістю, залишаються невирішеними, що вимагає постійних інновацій та адаптації відмовостійких механізмів для задоволення потреб цих систем, що постійно змінюються.

Досліджуючи ці реальні застосування, цей розділ демонструє практичну важливість відмовостійкості для забезпечення того, щоб розподілені системи могли продовжувати функціонувати правильно та безпечно, навіть за наявності збоїв. Надані візуальні підказки, включаючи графіки, таблиці та зображення, пропонують глибше розуміння того, як механізми відмовостійкості застосовуються в реальних системах, та проблем, які необхідно вирішувати в критичних середовищах.

2.4 Висновки по розділу

У другому розділі було досліджено методи та інструменти забезпечення відмовостійкості розподілених систем. Проведено класифікацію можливих типів відмов та визначено їх вплив на функціонування програмних систем. Розглянуто методи виявлення, локалізації та відновлення після збоїв, що забезпечують підтримку стабільності та безперервності роботи сервісів. Окрему увагу приділено технологіям реплікації даних, балансування навантаження та забезпечення високої доступності систем. У результаті дослідження визначено ефективні підходи до побудови fault-tolerant рішень у сучасних розподілених середовищах.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ FAULT-TOLERANT РІШЕНЬ

3.1 Платформи та інструменти для побудови відмовостійких систем

Хоча відмовостійкі механізми для розподілених систем реального часу в критично важливих додатках досягли значного прогресу, залишається кілька проблем і відкритих питань. Ці проблеми в основному пов'язані з підтримкою надійності системи, забезпеченням продуктивності в реальному часі, управлінням обмеженнями ресурсів і покращенням процесів відновлення після збоїв. Вирішення цих проблем має вирішальне значення для підвищення ефективності та стійкості відмовостійких систем у таких галузях, як аерокосмічна галузь, автономні транспортні засоби, промислове управління та охорона здоров'я. У цьому розділі досліджуються ці проблеми та представлені відкриті питання, які потребують подальших досліджень і розробок.

Однією з основних проблем у розподілених системах реального часу є балансування відмовостійкості з жорсткими часовими обмеженнями, які визначають програми реального часу. Системи реального часу повинні виконувати завдання в межах суворих часових обмежень, чого може бути важко досягти, коли відбуваються системні збої, що вимагає виявлення помилок, механізмів відновлення або виклику надлишкових компонентів. Впровадження механізмів відмовостійкості часто додає накладних витрат, що може вплинути на здатність системи дотримуватися цих термінів [39].

Накладні витрати на затримку: Багато механізмів відмовостійкості, такі як відновлення після помилок, міграція завдань або реплікація, вносять затримку. У критично важливих системах будь-яка додаткова затримка може призвести до пропуску термінів або неоптимальної продуктивності системи.

Складність планування: відмовостійкі алгоритми планування повинні динамічно адаптуватися до системних збоїв та перепризначати завдання, зберігаючи при цьому пріоритет критично важливих операцій. Досягнення цієї адаптивності без порушення часових обмежень залишається ключовою сферою

досліджень.

Відкриті питання:

Відновлення після збоїв з низькою затримкою: Розробка методів для швидшого виявлення та відновлення після збоїв, які мінімізують затримку та забезпечують продуктивність у режимі реального часу.

Передбачувані накладні витрати: створення механізмів відмовостійкості з мінімальними, передбачуваними накладними витратами, що дозволяє системам реального часу дотримуватися термінів навіть за умов збоїв.

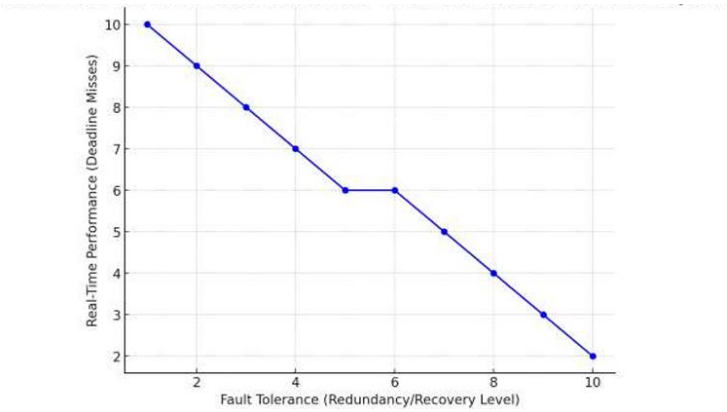

Рисунок 3.1 - Графік відображає компроміс між відмовостійкістю та продуктивністю в реальному часі

Графік показує компроміс між відмовостійкістю (з точки зору рівня резервування або відновлення) та продуктивністю в реальному часі (з точки зору пропусків термінів) у розподілених системах. Зі збільшенням відмовостійкості (більше механізмів резервування або відновлення) продуктивність у реальному часі має тенденцію до зниження, що призводить до більшої кількості пропусків термінів. Це підкреслює баланс, який системи повинні знаходити між забезпеченням надійності та своєчасним реагуванням.

Зі зростанням масштабу розподілених систем - чи то з точки зору кількості вузлів, завдань, чи географічного поширення - складність підтримки відмовостійкості зростає. Забезпечення ефективного масштабування відмовостійких механізмів без надмірних накладних витрат є критично важливим завданням.

Накладні витрати на зв'язок: Зі збільшенням кількості вузлів у розподіленій системі також збільшується обсяг зв'язку, необхідний для виявлення несправностей, відновлення та досягнення консенсусу. Цей додатковий зв'язок може призвести до затримок та неефективності.

Синхронізація станів: Підтримка узгодженості та синхронізації між зростаючою кількістю вузлів, особливо коли дії з відновлення повинні бути скоординовані, створює значні труднощі у великомасштабних системах.

Відкриті питання:

Масштабоване виявлення та відновлення несправностей: методи, які можуть масштабуватися до великих розподілених систем, мінімізуючи споживання ресурсів та накладні витрати на зв'язок.

Розподілений консенсус для відмовостійких систем: масштабовані та ефективні протоколи консенсусу, що гарантують, що навіть великі системи можуть підтримувати узгодженість без значних накладних витрат.

Ефективне управління ресурсами має вирішальне значення для забезпечення ефективної роботи відмовостійких систем без споживання надмірних обчислювальних ресурсів. У багатьох системах реального часу ресурси обмежені, а надмірне споживання ресурсів через відмовостійкі механізми може негативно вплинути на продуктивність системи.

Ключові питання:

Конкуренція за ресурси: У розподіленій системі такі ресурси, як процесор, пам'ять і пропускна здатність мережі, можуть використовуватися кількома завданнями. Механізми стійкості до відмов, такі як міграція завдань, реплікація або відновлення після помилок, можуть загострити конкуренцію за ресурси, що призведе до зниження продуктивності

Таблиця 3.1

Порівнюється масштабованість різних відмовостійких механізмів

Механізм	Сильні сторони	Обмеження	Масштабованість
Redundancy (Надмірність)	Висока відмовостійкість, простота впровадження.	Витрати на зберігання та мережеві ресурси.	Масштабується до певної межі; продуктивність падає при великій кількості реплік.
Paxos	Строга узгодженість, перевірено в системах.	Складність, високі витрати на зв'язок.	Погана масштабованість при великій кількості вузлів.
Raft	Легше впровадити, ефективний вибір лідера.	Менша відмовостійкість в екстремальних випадках.	Масштабується краще за Paxos, але лідер стає «вузьким місцем».
Quorum-Based	Гнучкий, відмовостійкий.	Падіння продуктивності при затримках і поділі мережі.	Висока масштабованість, але вразливий у розділених мережах.
Viewstamped Replication	Висока доступність, швидка зміна лідера.	Більше комунікацій для вибору лідера.	Масштабований у реальному часі, але чутливий до розділення мережі.

Наприклад, резервування, консенсусні алгоритми у великомасштабних розподілених системах, виділяючи їхні сильні та обмежені сторони.

Енергоефективність: У системах з обмеженими ресурсами, таких як пристрої Інтернету речей або автономні транспортні засоби, споживання енергії є критичним фактором. Стратегії відмовостійкості повинні збалансувати надійність системи з енергоефективністю.

Відкриті питання:

Оптимізований розподіл ресурсів: проектування відмовостійких систем, які можуть ефективно розподіляти ресурси навіть за умов збоїв, щоб уникнути конфлікту за ресурси.

Енергоефективна відмовостійкість: розробка відмовостійких механізмів, які мінімізують споживання енергії, особливо в системах з живленням від батарей або з обмеженими ресурсами.

Розподілені системи реального часу часто повинні обробляти кілька збоїв, що виникають одночасно, таких як збої обладнання, мережевих розділів

або програмних помилок. Наявність кількох одночасних збоїв ускладнює виявлення помилок, відновлення та реконфігурацію системи, особливо у великомасштабних системах.

Ключові питання:

Кореляція збоїв: Кілька збоїв можуть бути не незалежними та взаємопов'язаними, що може ускладнити визначення першопричини та вжиття відповідних заходів з відновлення.

Відновлення після одночасних збоїв: Забезпечення можливості відновлення системи після кількох збоїв без подальшого погіршення роботи або збоїв у застосунках реального часу.

Відкриті питання:

Діагностика відмов: Покращення здатності відмовостійких систем виявляти та діагностувати численні, корельовані збої в режимі реального часу.

Відновлення після кількох збоїв: Розробка алгоритмів та стратегій, які дозволяють системам відновлюватися після одночасних збоїв, водночас відповідаючи вимогам продуктивності в режимі реального часу.

Однією з класичних проблем у розподілених системах є компроміс між узгодженістю та доступністю, особливо у відмовостійких системах. У деяких випадках підтримка високої узгодженості під час відновлення після збоїв може поставити під загрозу доступність системи, і навпаки. Цей компроміс стає особливо вираженим у застосунках реального часу, де критично важливі як узгодженість, так і доступність.

Ключові питання:

Узгодженість даних: забезпечення узгодженого відображення даних усіма вузлами в розподіленій системі, особливо у разі збою або відновлення вузлів. У деяких системах забезпечення суворих гарантій узгодженості під час збоїв може призвести до затримок у відновленні.

Доступність: З іншого боку, забезпечення доступності під час збоїв, дозволяючи вузлам працювати із застарілими або невідповідними даними, може поставити під загрозу цілісність системи.

Відкриті питання:

Протоколи узгодженості: Розробка відмовостійких механізмів, які можуть забезпечити як узгодженість, так і доступність без шкоди для продуктивності системи в застосунках реального часу.

Адаптивна відмовостійкість: розробка адаптивних механізмів відмовостійкості, які можуть динамічно збалансувати узгодженість та доступність на основі критичності програми та умов відмови.

Безпека є ще однією важливою проблемою у відмовостійких розподілених системах, особливо в критично важливих сферах, таких як охорона здоров'я, аерокосмічна галузь та фінанси. Відмовостійкі механізми, що включають реплікацію, виявлення помилок або відновлення, можуть створювати вразливості, якщо вони не захищені належним чином, що призводить до потенційних порушень безпеки.

Ключові питання:

Зловмисні атаки: Механізми відмовостійкості можуть бути вразливими до атак, спрямованих на процеси відновлення або алгоритми консенсусу, таких як візантійські збої або атаки типу «відмова в обслуговуванні».

Цілісність та конфіденційність даних: забезпечення того, щоб механізми стійкості до відмов не порушували цілісність або конфіденційність конфіденційних даних під час процесів відновлення.

Відкриті питання:

Захист відмовостійких протоколів: Розробка безпечних відмовостійких механізмів, які захищають від зловмисних атак, зберігаючи при цьому надійність системи та її продуктивність у режимі реального часу. Захист даних під час відновлення після збоїв: Забезпечення того, щоб процеси відновлення ненавмисно не наражали конфіденційні дані на несанкціонований доступ.

Хоча механізми стійкості до відмов були успішно впроваджені в розподілених системах реального часу, вирішення цих проблем залишається постійним процесом. Необхідність підтримки продуктивності в реальному часі, забезпечення масштабованості системи та балансування компромісу між узгодженістю та доступністю є центральною для майбутніх досліджень [40]. Крім того, вирішення проблем управління ресурсами, обробка кількох

одночасних збоїв та підвищення безпеки під час відновлення після збоїв є ключовими областями, які потребують подальшого дослідження.

У цьому розділі окреслено основні проблеми та відкриті питання, що забезпечує всебічний огляд перешкод, які необхідно подолати для розробки більш стійких та ефективних відмовостійких систем для розподілених застосунків реального часу. Використання візуальних посібників, таких як графіки, таблиці та зображення, покращує розуміння цих проблем та їхнього впливу на продуктивність системи.

3.2 Архітектурні підходи до організації розподілених сервісів

Було розроблено та реалізовано кілька механізмів відмовостійкості у вигляді метаоб'єктних класів, реалізованих на FTS: механізм, заснований на протоколах стабільного сховища, первинного резервного копіювання та реплікації лідер-слідувач. У більшості випадків, з точки зору програміста відмовостійкості, розробка метаоб'єктних класів виконується з використанням одного й того ж шаблону класів. На рисунку 3.2 показано спрощений вигляд lfr_cmo . та LFR_SMO класи, що реалізують стратегію лідер-послідовник.

LFR_CMO	LFR_SMO

class LFR_CMO {
protected:
LFR_SMO FT_server;
public:
void Meta_StartUp () {...}
void Meta_MethodCall (...) {
...
reply = FT_server.FT_method_call (...);
...
}
};
	class LFR_SMO {
	protected:
	IRP_LFR_SMO Followers;
	public:
	void Meta_StartUp () {...}
	Argpac FT_method_call (...) {
	...
	Meta_HandleMethodCall(...);
	...
	Followers.FT_notify (...);
	}
	void FT_notify (...) {...}
	void FT_recover (...) {...}
	};

reflect LFR_SMO: CLIENT_MO;	reflect LFR_SMO: SERVER_MO; reflect IRP_LFR_SMO:CLIENT_MO;

Рисунок 3.2 - Класи метаоб'єктів відмовостійкості

Клас клієнта визначає, головним чином, як обробляється виклик методу: Meta_MethodCaii ( ) . Виклик методу поширюється на метаоб'єкт сервера за допомогою простого оператора: FT_server.FT_method_cail (). Метаоб'єкт сервера прозоро викликається з метаоб'єкта клієнта завдяки використанню верхнього метарівня розподілу. Сервер оголошується як локальний атрибут FT_server .

класу LFR_SMO . Цей клас пов'язаний з CLIENT_MO. обробка поведінки клієнта на метарівні дистрибуції. Клас сервера містить методи для обробки виклику (у цьому випадку всі репліки виконують метод) та протокол між репліками (IRp). Метод FT_method_call ( ) відповідає за обробку виклику методу сервера на базовому рівні.

Це робиться за допомогою Meta_HandleMethodCall ( ) Метод FT_notify Метод дозволяє лідеру синхронізуватися з послідовниками, повідомляючи їм про виконання певного методу. Це робиться після виконання будь-якого методу базового рівня за допомогою Followers.FT_notify ( ). Послідовники викликаються прозоро від лідера, знову ж таки завдяки використанню метарівня розподілу. У всіх механізмах реплікації збій репліки виявляється FTS, що активує FT_recover . метод однієї з активних реплік. У наведеному тут прикладі, якщо лідер виходить з ладу, то FT_recover змушує послідовника стати лідером та створення нової репліки у відповідному домені реплікації.

Два обов'язкові оголошення: reflect LFR_SMO : SERVER_MO та відобразити IRP_LFR_SMO : CLIENT_MO вказують на те, що сервер і послідовники відповідно пов'язані з метаоб'єктом сервера та метаоб'єктом клієнта на метарівні розподілу. Остання декларація дозволяє лідеру прозоро викликати послідовників під час протоколу міжреплікації, як уже згадувалося. Це звільняє метаоб'єкти відмовостійкості від обробки проблем розподілу, таких як віддалене створення/видалення, управління групами та атомарна передача багатоадресних повідомлень. Під час реалізації метаоб'єктів для протоколів реплікації програміст лише припускає, що всі Репліки серверів отримують однакові запити на виклик у однаковому порядку, навіть коли сервер спільно використовується кількома клієнтами. У цьому прикладі не використовується рівень безпечного зв'язку, тому метаоб'єкти стійкості до відмов пов'язані з метаоб'єктами розподілу. Якщо такий рівень використовується, то оголошення зв'язування необхідно відповідно оновити.

Введення метаоб'єктів для безпечного зв'язку ілюструє гнучкість, яку забезпечує метаоб'єктний підхід, описаний у цій роботі. Інтерфейс класів

метаоб'єктів на цьому рівні подібний до інтерфейсу класів, визначеного в попередньому розділі, як показано на рисунку 3.3.

SC_CMO

```
class SC_CMO {
protected:
 SC_SMO SC_server;
 session_key SK;
public:
 void Meta_StartUp () {
   // get SK from the
   // authentication server
   SC_server.SC_GetSessionKey(SK);
   ...
 }
 void Meta_MethodCall (...) {
   ...
   sign (request, SK);
   reply=SC_server.SC_method_call(...);
   check (reply, SK);
   ...
 }
};
```

```
reflect SC_SMO: CLIENT_MO;
```

SC_SMO

```
class SC_SMO {
protected:
 session_key SK;

public:
 void Meta_StartUp () {...}
 void SC_GetSessionKey (...) {...}
 ArPac SC_method_call (...) {
       ...
       check (request, SK);
       Meta_HandleMethodCall(...);
       sign (reply, SK);
       ...
 }
};
```

```
reflect SC_SMO: SERVER_MO;
```

Рисунок 3.3 - Класи метаоб'єктів безпечного зв'язку

На основі цієї моделі реалізовано класи метаоб'єктів, що відповідають за автентифікацію користувача клієнта. Наразі автентифікація базується на протоколі Нідхема-Шредера під час створення клієнта, а підписи використовуються під час кожного виклику сервера.

На стороні клієнта, Meta_startup автентифікує клієнта та отримує ключ сеансу. Цей ключ сеансу прозоро поширюється (у квитку) на сервер під час виклику.

SC_сервер . SC_GetSession -Key( ) . У Meta_MethodCall кожен виклик методу сервера підписується та прозоро поширюється на сервер за допомогою SC_server.SC_method_call () . На стороні сервера, SC_method_call перевіряє підпис і, якщо він правильний, виклик поширюється на базовий рівень за допомогою Meta_HandleMethodCall () . Базовим рівнем може бути рівень відмовостійкості або рівень програми.

Об'єктно-орієнтована розробка метаоб'єктів

У цьому розділі ми опишемо проектування механізмів відмовостійкості та підкреслимо, як метаоб'єкти забезпечують чітку та зручну основу для процесу проектування. Це головним чином тому, що об'єкти програми та метаоб'єкти відмовостійкості взаємодіють через чітко визначений інтерфейс, тобто протокол метаоб'єктів. Ця взаємодія автоматично обробляється під час компіляції. Крім

того, метаоб'єкти відмовостійкості проектуються та реалізуються без прямих операторів зв'язку, оскільки вони приховані в метаметаоб'єктах зв'язку. Таким чином, проектування метаоб'єктів відмовостійкості може здійснюватися без урахування ні функціональності програми, ні деталей зв'язку.

Спочатку ми коротко представимо графічну нотацію, яка використовується для опису деяких етапів нашого проектування. Ця нотація взята з BON (Business Object Notation) і забезпечує підтримку для опису як структури, так і поведінки системи. Статичні діаграми описують структуру системи з точки зору класів, представлених трикрапками, та їх зв'язків, успадкування або складу, відповідно представлених одинарною та подвійною стрілками. Динамічні діаграми представляють поведінку системи з точки зору повідомлень, якими обмінюються об'єкти. Об'єкти представлені прямокутниками та названі за їхнім класом; якщо це недостатньо точно, також згадується ім'я екземпляра. Повідомлення представлені пунктирними стрілками з порядковим номером та орієнтовані від відправника до одержувача. З кожною динамічною діаграмою пов'язаний сценарій; це таблиця, де наведено опис кожного повідомлення відповідно до його порядкового номера. Стрілки, що не мають об'єкта походження, представляють зовнішні вхідні події, отримані системою, а стрілки, що не мають об'єкта призначення, представляють зовнішні вихідні події.

При зворотному відновленні після помилок безпомилковий стан замінює виявлений помилковий стан як такий; це перетворення стану полягає у поверненні системи до попереднього правильного стану. Це включає визначення точок відновлення, які є моментами часу під час виконання процесу, для яких поточний стан може згодом потребувати відновлення.

Об'єктна модель та використання протоколів метаоб'єктів заохочують визначення точок відновлення в кінці виконання методу. Коли виявляється помилка, поточний запущений метод має бути повторно виконаний з самого початку, що передбачає відновлення початкових умов (зокрема , стану об'єкта базового рівня) та ідентифікацію поточного запущеного методу (номер методу, аргументи тощо). Це забезпечується класами FT_CMO. та FT_SMO , перший з яких

реалізує клієнтську частину протоколу відмовостійкості, а другий - серверну. Виклики методів рівня програми перехоплюються у FT_CMO. методом Meta_MethodCaii який передає виклик базового методу до FT_SMO . Виконання виклику клієнтського методу відбувається у методі FT_method_call. з FT_SMO метаоб'єкт.ft_smo виконує запитуваний метод та визначає точку відновлення перед тим, як відправити результат виконання методу назад до FT_CMO . Обидва класи є успадкованими та спеціалізованими. для всіх механізмів, описаних нижче. Протокол метаоб'єктів також надає засоби для доступу до атрибутів об'єкта з метарівня. Інформація про стан тут розглядається як набір атрибутів об'єкта.

Інтерфейси метаоб'єктів відмовостійкості, наведені на рисунку 3.4, є узагальненням метаоб'єктів реплікації лідер-послідовник. Інтерфейс метаоб'єкта клієнта в основному полягає в перевизначенні Meta_MethodCall. (що належить до MetaObj), щоб він викликав FT_method_caii на метаоб'єкті сервера. Цей метаоб'єкт сервера відображається локально у FT_CMO через проксі-сервер під назвою FT_server . Той факт, що він віддалений, стає прозорим завдяки використанню оголошень reflect на стороні клієнта та сервера. Оголошення reflect FT_SMO : CLIENT_MO партнери FT_сервер (екземпляр FT_SMO) з метаоб'єктом зв'язку клієнтського типу, який перетворює FT_server у проксі-сервер, щоб віддалений метод FT_method_call можна виконати за допомогою простого виклику методу FT_ server.FT оператор. На стороні сервера, оголошення FT_SMO : SERVER_MO робить FT_SMO наприклад, сервер , подібний до RPC, який очікує запитів від віддалених клієнтів. Ініціалізація стеку та реєстрація у відповідній групі служб виконується шляхом перевизначення Meta_startup метод.

```
class FT_CMO: public MetaObj {
  FT_SMO FT_server;
public:
  void Meta_StartUp ();
  void Meta_MethodCall (...);
};
```

```
class FT_SMO: public MetaObj {
public:
  void Meta_StartUp ();
  ArgPac FT_method_call (...);
  void FT_method_end () = 0;
  void FT_recover () = 0;
};
```

Рисунок 3.4 - Базовий інтерфейс для механізмів зворотного відновлення

У FT_method_call , Meta_HandleMethodCall (також належить до MetaObj) викликається першим , щоб поширити виклик методу на базовий рівень; потім FT_method_end викликається для визначення точки відновлення. Той факт , що ефективне виконання методу обробляється MetaObj Виклик методу обробки мета-обробника є важливим, оскільки це означає, що програмісту відмовостійкості не потрібно явно викликати метод рівня програми і, отже, не потрібно знати про функціональність програми. FT_recover автоматично викликається базовою службою системи виявлення помилок, коли виявляється помилка. Обидва методи сильно залежать від механізму і тому є абстрактними методами, що робить FT_SMO абстрактний клас.

У цьому механізмі поточний виклик (ідентифікатор методу та аргументи) та стан сервера зберігаються у стабільному сховищі в кінці кожного виклику методу. Помилки сигналізуються метаоб'єкту клієнта , а відновлення полягає у виборі дійсного сайту, створенні сервера-замінника на цьому сайті та, за необхідності , повторній надсиланні поточного виклику. Ініціалізація сервера-замінника полягає у відновленні останнього виклику та стану, доступного у стабільному сховищі.

Тут ми представляємо два класи, що взаємодіють із системними сервісами. STABLE_STORAGE надає методи зчитування та запису даних у стабільне сховище (NFS у наших експериментах) у формі ArgPac екземпляри. DOMAIN зберігає узгоджений вигляд дійсних сайтів, взаємодіючи зі службою виявлення помилок у підсистемі відмовостійкості (xAMp у наших експериментах). Він надає методи для перевірки цього вигляду та отримання назв дійсних сайтів.

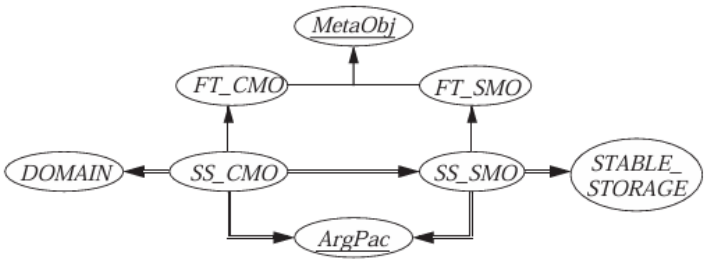

Рисунок 3.5 - Статична діаграма — Точки відновлення на стабільному сховищі

Виклик та стан реалізовано за допомогою Open C++ ArgPac попередньо визначений клас, який діє як контейнер метарівня для будь-якого типу аргументів. Класи SS_CMO та SS_SMO відповідно реалізувати клієнтську та серверну частини механізму та успадкувати відповідно від FT_CMO та FT_SMO . Ці метаоб'єкти причинно-наслідково пов'язані з метаоб'єктами групової комунікації, що дозволяє SS_SMO ЩОБ SS_CMO міг це побачити з точки зору моделювання , а також на практиці, через просте посилання на композицію. Це посилання приховує використання проксі-версії SS_SMO призначений для віддаленої взаємодії з SS_CMO та спрощує конструкцію. На рисунку 3.5 наведено статичну діаграму механізму на основі стабільного сховища.

Тепер ми розглянемо два важливі сценарії виконання: визначення точки відновлення в кінці виконання методу та відновлення після виявлення помилки. Перший реалізовано за допомогою FT_method_end. у SS_SMO (на стороні сервера) та викликає метод Save, що надається STABLE_STORAGE. Цей сценарій ілюструється на рисунку 3.6.

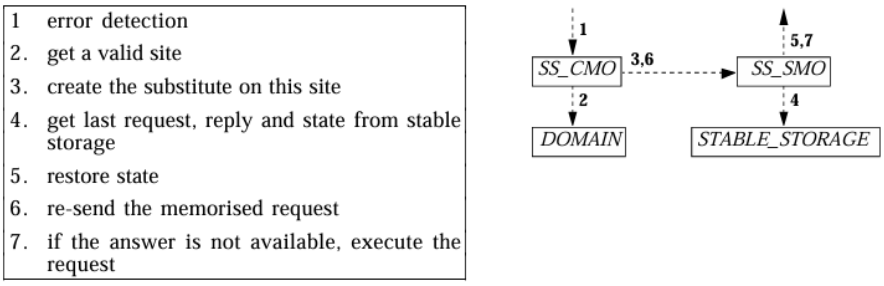

Рисунок 3.6 - Сценарій — Визначення точки відновлення

Визначення точки відновлення в основному полягає у захопленні базового стану та збереженні його у стабільному сховищі . Оскільки цей базовий стан стає доступним через протокол метаоб'єктів, програмісту відмовостійкості не потрібно викликати жодної специфічної для програми процедури.

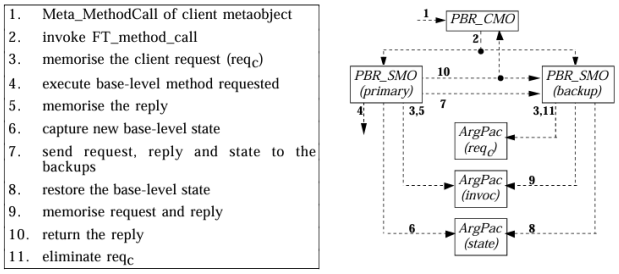

Рисунок 3.7 - Сценарій - Визначення точки відновлення

Другий сценарій описує процедуру відновлення. Під час виконання методу помилка може бути виявлена службою системи виявлення помилок або до збереження нового стану у стабільному сховищі, або між часом резервного копіювання та часом, коли сервер надсилає відповідь. В обох випадках помилка сигналізується метаоб'єкту клієнта, який запускає процедуру, описану на початку цього розділу та проілюстровану на рисунку 3.7. Що стосується захоплення стану, відновлення стану обробляється через протокол метаоб'єкта і тому не передбачає використання специфічних для програми операторів. Інформація про стан отримується за допомогою FT_SMO за допомогою Meta_HandleRead створюється нова репліка створено, а його стан відновлюється за допомогою Meta_HandleAssign .

У цій стратегії реплікації всі репліки сервера належать до однієї атомарної групи багатоадресної розсилки і, таким чином, отримують однакові вхідні повідомлення (наприклад, повідомлення виклику) в одному порядку. Це припущення не є обов'язковим, але спрощує проектування механізму. Воно забезпечується використанням протоколів багатоадресного зв'язку та груп на метарівні зв'язку. Серед реплік лише одна (основна) обробляє запити клієнта та встановлює контрольні точки свого нового стану в кінці кожного виконаного методу для інших реплік (резервних копій). Ми припускаємо, що будь-яка помилка основної копії виявляється резервними копіями, які обирають нову основну копію серед себе. Ця нова основна копія відновлює останній стан з контрольною точкою та, за необхідності, виконує поточний запит, перш ніж повернути відповідь клієнту. Ми представляємо PBR_CMO та PBR_SMO класи метаоб'єктів (похідні від FT_CMO та FT_SMO відповідно) для реалізації

клієнтської та серверної частин реплікації первинної резервної копії. Будь-яка резервна копія може стати первинною, тому як первинна, так і резервна поведінка реалізуються за допомогою PBR_SMO . Первинний або резервний стан PBR_SMO Метаоб'єкт обробляється через локальну змінну. Первинний об'єкт можна розглядати як клієнт резервних копій, а отже, PBR_SMO є клієнтом самого себе, як показано на рисунку 3.8. Взаємодія репліки (тобто взаємодія між PBR_SMO екземпляри) стає прозорим завдяки використанню метаоб'єктів зв'язку, як і для взаємодії метаоб'єктів клієнт/сервер.

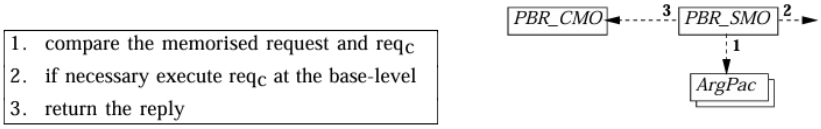

Рисунок 3.8 - Сценарій — Процедура відновлення

Сценарій визначення точки відновлення подібний до сценарію механізму стабільного сховища. Основна відмінність полягає в тому, що записаний стан та виклик передаються до резервних копій, а не зберігаються у стабільному сховищі. Це також можна зробити за допомогою деяких засобів протоколу метаоб'єктів і не вимагає жодного програмування, специфічного для програми, для програміста відмовостійкості.

Сценарій відновлення (рисунок 3.8) набагато простіший , ніж у випадку стабільного сховища, оскільки у випадку реплікації реконфігурація (тобто клонування нової репліки на валідному сайті) не є обов'язковою.

Виконання методу на базовому рівні має здійснюватися лише тоді, коли доступна відповідь не відповідає запам'ятованій *req* $_c$. Його обробляє Meta_HandleMethodCall і тому не потребує жодного програмування на рівні програми. PBR_SMO Екземпляр, який обробляє подальше відновлення, стає новим основним і, отже, відповідає за виконання наступних клієнтських запитів.

Цей механізм реплікації натхненний напівактивною реплікацією в Delta-4. У цьому протоколі всі репліки обробляють вхідні повідомлення, але лише одна

(лідер) надсилає вихідні повідомлення. Як і у випадку первинно-резервної реплікації, всі репліки сервера належать до однієї атомарної групи багатоадресної розсилки та отримують однакові повідомлення в одному порядку. У нашій реалізації лідер спочатку виконує запит, а потім повідомляє про нього інші репліки (послідовники), які, у свою чергу, виконують запит. Тільки лідер повертає відповідь клієнту. Класи метаоб'єктів, отримані для реалізації цього протоколу, називаються LFR_CMO. та LFR_SMO . Статична діаграма та сценарії досить схожі на ті, що отримані для реплікації первинної резервної копії .

Вищезазначені механізми мають кілька спільних структурних та поведінкових моментів. Їх можна факторизувати та повторно використовувати в ієрархії. У всіх цих механізмах існує резервна копія стану сервера, або на стабільному сховищі, або як стан репліки. Ми вводимо клас РЕЗЕРВНОГО КОПІЮВАННЯ , який використовується в будь-яких механізмах зворотного відновлення. Цей клас визначає FT_update . метод; це абстрактний метод, оскільки він залежить від реалізованого механізму. FT_CMO / FT_SMO фреймворк доповнено РЕЗЕРВНИМ КОПІЮВАННЯМ FT_SMO є клієнтом РЕЗЕРВНОГО КОПІЮВАННЯ (композиційне посилання). Усі раніше визначені класи STABLE_STORAGE , PBR_SMO та LFR_SMO успадковувати з РЕЗЕРВНОЇ КОПІЇ та визначати FT_update відповідно як:

- операція запису в стабільне сховище;
- передача нового стану основного сервера резервним серверам ;
- повідомлення про виконання методу послідовникам, які, у свою чергу, його виконують.

Що стосується стабільного сховища та реплікації основної резервної копії, захоплення стану систематично виконується після завершення методу перед оновленням резервної копії (файлу стабільного сховища або репліки резервної копії). Це також можна факторизувати у визначенні FT_method_end. у класі

КОНТРОЛЬНА ТОЧКА_SMO який УСПАДКОВУЄТЬСЯ ВІД FT_SMO.SS_SMO та PBR_SMO успадковувати від CHECKPOINT_SMO та розширити FT_method_end

відповідно, щоб зберегти захоплений стан у стабільному сховищі або надіслати його до групи резервних копій.

Для обох механізмів реплікації процедура відновлення та обробка клієнтських запитів запасними репліками (резервними копіями та послідовниками) є схожими. Цю поширену поведінку також можна факторизувати в класі REPLICA_SMO. успадкування від FT_SMO . REPLICA_SMO також успадковується від РЕЗЕРВНОЇ КОПІЇ , оскільки її екземпляри, що використовуються в запасних репліках, містять послідовно збережені стани. PBR_SMO та LFR_SMO УСПАДКОВУЄТЬСЯ від REPLICA_SMO ТА lfr_smo перевизначає FT_method_end щоб поточний виклик був повідомлений послідовникам. PBR_SMO множення успадковується від CHECKPOINT_SMO та REPLICA_SMO . Вони також обробляють реконфігурацію шляхом *клонування* нової репліки. Ці структурні та поведінкові спільні точки, факторизовані класи призводять до ієрархії класів, представленої на рисунку 3.9.

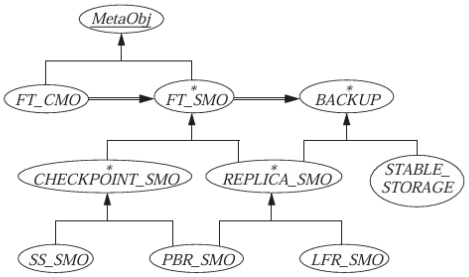

Рисунок 3.9 - Ієрархія класів метаоб’єктів зворотного відновлення

3.3 Оцінювання ефективності та продуктивності fault-tolerant систем

Простота використання. Механізми відмовостійкості можна додати до програми, просто підключивши відповідні метаоб'єкти до об'єктів програми (використовуючи кілька операторів оголошення метаоб'єктів). Крім того, оскільки всі метаоб'єкти відмовостійкості мають однаковий інтерфейс (MetaObj інтерфейс) механізм, що використовується, можна змінити простим перемиканням метаоб'єктів. Таким чином, програміст додатків може розробляти об'єкти додатків локально як єдиний блок середовища виконання та тестувати їх

функціональну поведінку. Після того, як ця перша версія запрацює належним чином, її можна розповсюджувати, підключаючи віддалені об'єкти до метаоб'єктів зв'язку, за умови, що кожен основний об'єкт додатка перетворюється на єдиний блок середовища виконання. Потім, наприклад, можна вставити метаоб'єкти стабільного сховища, щоб зробити додаток відмовостійким. З міркувань продуктивності, наприклад, накладних витрат часу відновлення, програміст додатків може видалити метаоб'єкти стабільного сховища та замінити їх метаоб'єктами типу "лідер-підпорядник". Усі ці операції полягають лише у з'єднанні об'єктів додатків та метаоб'єктів і не передбачають суттєвих змін в об'єктах додатків.

Загальність. Цей аспект насправді не пов'язаний з нашою моделлю програми, а радше з самою природою реалізованих механізмів, які можна використовувати в широкому спектрі програм. Коротко кажучи, використання об'єктно-орієнтованого методу проектування та протоколу метаоб'єктів дозволяє вбудовувати та адаптувати механізми в об'єкти програми без будь-яких змін у вихідний код програми. Адаптація може відбуватися шляхом виведення нових механізмів з існуючих, але взаємодія між об'єктами програми та метаоб'єктами фіксується через МОР.

Розширюваність. Нові механізми також можуть бути похідні від існуючих шляхом успадкування на метарівні відмовостійкості. Наприклад, ми можемо похідні від існуючих механізмів, заснованих на дельта-контрольних точках та/або реєстрації повідомлень, або механізми, де стан резервується лише після виконання методу запису, або ж різні стратегії активної реплікації. Ці механізми потім можна використовувати так само легко, як і інші, шляхом простого причинно-наслідкового зв'язку з об'єктами програми. Більш проблематичним є розширення механізмів при зміні припущень, зроблених щодо властивостей зв'язку . Припустимо, що на заданій платформі відсутній протокол управління групами та багатоадресної розсилки, і потрібна розробка протоколу реплікації, подібного до первинного резервного копіювання. Тоді, хоча й можливо, повторне використання поточних метаоб'єктів первинного резервного копіювання може бути обмеженим, але це не пов'язано з підходом до

метаоб'єктів.

Композитність. Різні типи механізмів відмовостійкості можна використовувати в одній програмі шляхом з'єднання різних віддалених об'єктів з різними типами метаоб'єктів. Наприклад, клієнтський об'єкт може взаємодіяти з віддаленим об'єктом, що має стабільне сховище, та з іншим віддаленим об'єктом, реплікованим відповідно до моделі первинної резервної копії. Використання кількох метарівнів також забезпечує композитність між різними метафункціональними властивостями. Це лише питання оголошень під час компіляції. Не потрібно вносити жодних змін до вихідного коду метаоб'єкта відмовостійкості, коли використовуються лише метаоб'єкти розподілу або коли також використовуються метаоб'єкти безпеки. Метаоб'єкти безпеки можна вставляти в нашу архітектуру між метаоб'єктами відмовостійкості та групового зв'язку.

Багаторівневий підхід дозволяє апріорі ігнорувати групові комунікації , оскільки вони забезпечуються (наступним) метарівнем. Це не означає, що властивості групової комунікації ігноруються під час розробки метаоб'єктів. Це ключовий аспект проектування метаоб'єктів відмовостійкості, який необхідно враховувати, оскільки властивості системи комунікації спрощують рішення для відмовостійкості.

Однак, реалізація розподілених механізмів відмовостійкості передбачає сильну взаємодію з груповими комунікаційними службами. Наприклад, під час відновлення, комунікаційні метаоб'єкти повинні очищати деякі черги повідомлень. Саме тому проектування комунікаційних метаоб'єктів має... враховувати деякі аспекти стратегій відмовостійкості. Це також стосується метаоб'єктів безпеки, наприклад, поширення ключів сеансу на нову репліку під час відновлення має виконуватися метаоб'єктами безпеки. Це означає, що під час проектування метаоб'єктів необхідно вживати запобіжних заходів; контекст використання має бути точно визначений, а роль супутніх метаоб'єктів має бути врахована; тим не менш, з практичної точки зору, програміст відмовостійкості не відповідає за обробку викликів до підсистеми групового зв'язку, а лише бере участь в оголошеннях зв'язків між об'єктами.

Ця архітектура дозволяє використовувати інші групові комунікаційні служби за умови, що вони пропонують ті самі комунікаційні властивості, наприклад, атомарну багатоадресну розсилку повідомлень виклику. Інтерфейс до базової комунікаційної системи може бути іншим. Якщо властивості, що надаються комунікаційною системою, слабші (наприклад, глобальне впорядкування доставки повідомлень не передбачено, як для груп Chorus), то це сильно впливає на проектування метаоб'єктів відмовостійкості, оскільки більше не можна передбачати ідентичну доставку вхідних повідомлень. Також для метаоб'єктів безпеки вплив може бути пов'язаний з властивостями комунікаційної підсистеми щодо безпеки; якщо повідомлення шифруються апаратними пристроями на дуже низькому рівні, то забезпечення конфіденційності передачі повідомлень більше не залежить від використання метаоб'єктів.

У цій архітектурі структурування базується на стеку метаоб'єктів, і таким чином виклики об'єктів рекурсивно перехоплюються метаоб'єктами у стеку на стороні клієнта та ефективно виконуються метаоб'єктами у стеку на стороні сервера. Фактично, одна й та сама модель програмування використовується на будь-якому рівні, як на рівні програми, так і на будь-якому метарівні. До будь-якого віддаленого об'єкта звертається проксі-сервер в адресному просторі клієнта. Протокол між проксі-об'єктом та віддаленим об'єктом обробляється парою метаоб'єктів (клієнт і сервер). Ця проста модель програмування дуже зручна, оскільки вона добре зрозуміла як програмістам програм, так і системним програмістам. Наприклад, на метарівні відмовостійкості група реплік сервера розглядається як єдиний локальний об'єкт, "резервна копія", метаоб'єктом клієнта. "Група резервних реплік" розглядається як унікальний локальний об'єкт основною реплікою метаоб'єкта. Цей об'єкт пов'язаний з метаоб'єктом, що обробляє зв'язок у "групі резервних копій". На метарівні безпеки сервер автентифікації також розглядається як унікальний та локальний об'єкт.

Порядок метарівнів може змінюватися залежно від властивостей, які необхідно гарантувати. За винятком останнього метарівня (відповідального за зв'язок) та базового рівня (об'єкт програми), перестановка всіх проміжних рівнів

може здаватися можливою та обґрунтованою. Хоча це цілком можливо з практичної точки зору (усі класи метаоб'єктів мають один і той самий інтерфейс протоколу метаоб'єктів), порядок фактично впливає на властивості, які повинна мати кінцева програма. Наприклад, якщо цілісність повідомлення має бути гарантована (наприклад, за допомогою обчислення та перевірки сигнатури) також для інформації, доданої під час викликів методів метарівнем відмовостійкості, тоді метарівень безпеки має бути викликаний після метарівня відмовостійкості. Якщо цілісність повідомлення має бути гарантована лише для інформації програми під час виклику методу (метод та параметри), тоді можна очікувати, що метарівень безпеки буде розміщено одразу після базового рівня. Припустимо, що проміжні рівні організовані в такому порядку: метарівень безпеки розміщується одразу після рівня програми, потім метарівень відмовостійкості, що призводить до наступної послідовності (A; SC; FT; GD). Коли первинний об'єкт займає контрольну точку об'єкта програми, доступ до стану базового рівня повинен проходити через метарівень безпеки. Це означає, що метарівень безпеки може поширювати цей доступ вниз до свого базового рівня, що неможливо з протоколом метаоб'єктів, який ми використовуємо. Оновлення резервного стану також було б проблемою, оскільки запис атрибутів базового рівня повинен проходити знову через метарівень безпеки. Це можна вирішити, незначно змінивши MOP, зробивши доступним як базовий, *так і* нижній (прикладний) рівень незалежно від розташування метаоб'єкта в стеку. Іншим рішенням було б реалізувати ланцюжок метаоб'єктів за допомогою плоского зв'язування замість стеку. З поточним рішенням (A; FT; SC; GD) та використаним простим MOP, читання або запис атрибутів базового рівня легко досягається. Більше того, помилки, виявлені метарівнем безпеки, такі як помилка автентифікації після кількох повторних спроб, можуть бути передані на базовий рівень безпеки, тобто метарівень відмовостійкості, який обробляє дії відновлення. Фактично, згідно з очікуваними властивостями програми, це єдина відповідна послідовність проміжних рівнів, якщо потрібні як відмовостійкість, так і безпечний зв'язок.

Метаоб'єкти відмовостійкості реалізують частину механізмів, головним

чином обробку помилок. Вони спираються на підсистеми, що реалізують сервіси, спільні для метаоб'єктів. У FRIENDS підсистеми, пов'язані з відмовостійкістю (FTS, GDS), відповідають за виявлення помилок, груповий зв'язок, управління доменами реплікації, клонування об'єктів під час реконфігурації, управління стабільним сховищем. Причини, чому ці сервіси реалізовані як підсистеми на мікроядрі, такі:

- деякі служби часто залежать від спеціалізованих апаратних компонентів (наприклад, сторожових таймерів для виявлення помилок, контролерів мережевого підключення без збоїв для групового зв'язку, подвійних плат пам'яті з автономним джерелом живлення для стабільного зберігання); відповідне програмне забезпечення повинно працювати на виділених компонентах і потребує доступу до адресного простору ядра (наприклад, актори Chorus, що працюють у режимі супервізора під час реалізації протоколів xAMP у FRIENDS).
- деяким іншим службам потрібен глобальний огляд розподіленої архітектури (наприклад, глобальний огляд доступних вузлів, стану та конфігурації для керування доменами реплікації) та віддалені засоби, що надаються мікроядром (наприклад, системний виклик мікроядра для віддаленого створення об'єктів під час реконфігурації).
- Розділення між метаоб'єктами та підсистемами також цікаве для повторного використання готових компонентів; у FRIENDS служба комунікації програмного забезпечення xAMp була повторно використана та реалізована як підсистема на Chorus, файлова система Unix була повторно використана як частина підсистеми Unix SCO, доступної на Chorus Fusion 2.0.

Усі сервіси, що надаються підсистемами, часто є обов'язковими для пов'язаних бібліотек метаоб'єктів. Вони не підлягають змінам, хоча метаоб'єкти, що реалізують точні механізми, можуть досить легко розвиватися.

Роль метаоб'єктів може змінюватися залежно від метафункціональної властивості, реалізованої на метарівні.

Відмовостійкість. Це вже було проілюстровано в попередніх розділах для толерантності до фізичних несправностей. Дійсно, метаоб'єкти для фізичної

відмовостійкості реалізують досить багато механізмів, навіть якщо вони покладаються на деякі обов'язкові базові служби. Це може бути інакше для інших типів несправностей, таких як програмні несправності. У цьому випадку більшість механізмів можна реалізувати як метаоб'єкти. Підсистеми та низькорівневий доступ до системного програмного забезпечення в цьому випадку марні. Різниця тут полягає в тому, що поведінка метаоб'єктів залежить від інформації, наданої об'єктами додатків, що насправді не було так під час обробки фізичних несправностей. Наприклад, при реалізації блоків відновлення або N-версійного програмування набір версій має бути оголошений програмістом додатків на метарівні. Підпрограми-аудитори також повинні бути оголошені, оскільки вони залежать від додатка. Підсумовуючи, рівень додатка повинен параметризувати метарівень. Це може вплинути на протокол метаоб'єктів.

Безпека. Забезпечення безпеки в розподіленій системі включає багато різних механізмів, структурування системи та базових властивостей. У FRIENDS ми розглядали лише автентифікацію, конфіденційність та цілісність передачі повідомлень, які, як виявилося, досить легко обробляються метаоб'єктами. Підсистема SCS тут відповідає за автентифікацію користувачів та управління ключами. Існування такої підсистеми дозволяє інтегрувати деяке програмне забезпечення для автентифікації COTS у FRIENDS, таке як Kerberos, наприклад. Ще однією потребою для цієї підсистеми SCS є те, що вона повинна відповідати за безпечне зберігання постійних ключів. Аспекти авторизації ще не розглядалися. Тим не менш, очевидно, що ці аспекти повністю залежать від глобальної інформації (права користувача) та структурування системи (TCB, ядро безпеки). В ідеалі, перевірка прав доступу відповідно до заданої політики безпеки (як для конфіденційності , так і для цілісності) повинна бути реалізована ядром безпеки, яке є частиною мікроядра. Крім того, це ядро безпеки має бути захищеним від несанкціонованого доступу, завжди викликатися та перевірятися правильно. Останні властивості нав'язують підходи до проектування та реалізації, які, на нашу думку, не мають нічого спільного з метаоб'єктами. Роль метаоб'єктів у цьому випадку полягає лише в перехопленні викликів об'єктів та

передачі цього виклику ядру безпеки для авторизації .

Короткий зміст. Головною метою цього підходу є максимізація механізмів, що обробляються в межах метаоб'єктів. Як наслідок , механізми можна легше оновлювати, оскільки вони частково реалізуються на рівні користувача. Також головною метою є ефективна реалізація метафункціональних властивостей . Саме тому метаоб'єкти та супутні підсистеми на мікроядрі здаються перспективним рішенням. Роль метаоб'єктів у реалізації метафункціональних властивостей може сильно відрізнятися; варто зазначити, що для деяких з них роль метаоб'єктів не є суттєвою (лише додаткова непряма інформація). З абстрактної точки зору, метаоб'єкти та відповідна підсистема (включаючи мікроядро) можна розуміти як реалізацію заданого метарівня. Хоча з концептуальної точки зору підсистеми є частиною метарівня, мотивацією для цієї реалізації було скористатися перевагами як архітектури метарівня (легке налаштування програм щодо нефункціональних вимог), так і технології мікроядра (легке налаштування ОС).

Обмеження та недоліки. Ті, що були виявлені досі, по суті стосуються використаного МОР. Щоб подолати проблему, пов'язану зі статичним зв'язуванням класів додатків з класами метаоб'єктів, класам додатків, що визначають однакову поведінку, необхідно надати кілька назв. Крім того, Open C++ V1 надає обмежену метаінформацію, і тому об'єкти додатків не були реалізовані за допомогою успадкування. Однак останнє обмеження можна послабити за допомогою потужнішого МОР, що забезпечує контроль над деревом успадкування. Інший момент стосується абстракцій, з якими мається на увазі. Організація стеку метаоб'єктів можлива, оскільки ні рівень безпеки, ні рівень зв'язку не потребують доступу до рівня додатків. Це нетипово для цих абстракцій, а інші, такі як аспекти м'якого реального часу, можна легко обробляти в цьому стеку. Тим не менш, якщо двом метарівням потрібно отримати доступ до рівня додатків, то їх відповідна поведінка повинна оброблятися на одному метарівні. Тут знову ж таки це обмеження можна послабити за допомогою багатшого МОР, що дозволяє систематичний доступ як до базового, так і до нижнього базового рівня. З точки зору проектування

метаоб'єктів стійкості до відмов також важливо враховувати властивості системи зв'язку.

Ще один урок, який ми засвоїли з цього досвіду, полягає в тому, що зміна порядку рівнів у стеку , коли це можливо, призводить до різних властивостей. Це необхідно ретельно проаналізувати при використанні кількох метарівнів, оскільки такі зміни можуть призвести до неочікуваних побічних ефектів, як показано вище. Кілька інших проблем також нелегко вирішити, але, на нашу думку, вони не є такими через використання метаоб'єктів (обробка кількох відповідей, зміни перегляду, чи слід вважати їх збоями тощо). Нарешті, система FRIENDS сьогодні дуже залежить від мови програмування (Open C++) та від її власної об'єктно-орієнтованої розподіленої підтримки. Альтернативний підхід, що використовує операційну реалізацію МОР в межах об'єктно-орієнтованого рівня виконання, був би незалежним від мови. CORBA-сумісні шари та COTS (включаючи мікроядра) будуть розглянуті найближчим часом .

Динамічне зв'язування метаоб'єктів не було реалізовано у FRIENDS сьогодні. Навіть якщо динамічне зв'язування може ускладнити перевірку, це можливо за допомогою рефлексивного мовного підходу. У цьому випадку об'єкт та метаоб'єкти є окремими сутностями середовища виконання. Будь-який об'єкт програми має локальний простий метаоб'єкт (на основі класу MetaObj в Open C++), який перенаправляє дії, що мають бути виконані, до ефективного метаоб'єкта за допомогою локальних механізмів зв'язку. Інтерфейс програмування для програміста додатків залишається тим самим. Реалізація метаоб'єктів (зокрема відновлення) у цьому випадку має бути переглянута через необхідність (і) доступу до інформації про стан базового рівня, (ii) обміну інформацією між метаоб'єктами.

Як показано в прикладі, написання застосунків на FRIENDS по суті призводить до написання традиційного коду C++. Єдиним додатковим кодом, який потрібен, є оголошення МОР reflect, яке з'єднує об'єкт з метаоб'єктом. На практиці можна використовувати кілька метаоб'єктів, як пояснюється в роботі. Отже, коли клієнт-серверна програма вимагає лише розподілу, то перше та унікальне оголошення відповідає оголошенню метаоб'єкта розподілу як у

клієнті, так і на сервері. Коли потрібна відмовостійкість, то це оголошення замінюється оголошенням метаоб'єктів відмовостійкості. Метаоб'єкти розподілу в цьому випадку оголошуються в метаоб'єктах відмовостійкості (див. рис. 3.10). З практичної точки зору, ці оголошення можуть бути автоматично включені в програму за допомогою автономного інструменту конфігурації та можуть бути оновлені без будь-якого втручання користувача під час компіляції.

необхідно дотримуватися кількох правил програмування . Наприклад, для будь-якого класу об'єктів C, препроцесор Open C++ генерує рефлективний клас (refl_C), коли C пов'язаний з класом метаоб'єктів. Рефлективний клас має використовуватися замість стандартного під час створення рефлективних об'єктів. Оскільки Open C++ V1 MOP не підтримує успадкування, слід уникати похідних класів . Нарешті, будь-який рефлективний об'єкт сервера FRIENDS успадковує метод init для ініціалізації та метод start. Останній метод активує цикл отримання повідомлень на метарівні зв'язку.

На рисунку 3.10 показано вплив зміни швидкості обміну повідомленнями між процесами програми на накладні витрати на безвідмовність програми. Збільшення швидкості обміну повідомленнями головним чином впливає на час, витрачений на доповнення *кожного* повідомлення бухгалтерською інформацією (12 байтів, що містять: інтервал станів, ідентифікатор відправника, порядковий номер надсилання) та відстеження залежностей обмінюваних повідомлень, тобто ведення записів про останнє надіслане та отримане повідомлення для кожної програми. Ці накладні витрати є постійними, поки між процесами програми надсилаються/отримуються повідомлення, тоді як накладні витрати на реєстрацію повідомлень впливають на виконання програми лише в тому випадку, якщо повідомлення перетинає лінію відновлення через нашу політику вибіркового реєстрування повідомлень. Згодом накладні витрати на безвідмовність поступово зростають, але навіть за значної швидкості обміну повідомленнями (6,4 Кбайт/с) накладні витрати були виміряні на рівні *2,9%* , що зручно потрапляє в загальноприйнятні *"10%"* накладних витрат на час виконання програми [14].

Сенс у [13] реалізував незалежну схему контрольних точок з песимістичним веденням журналу повідомлень, і для значно менш вимогливого до зв'язку застосунку (0,06 Кбайт/с) накладні витрати становили *3,0%* з аналогічним інтервалом контрольних точок *2 хвилини.*

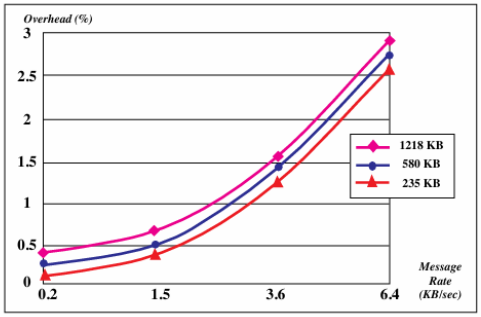

Рисунок 3.10 - Накладні витрати на безвідмовність як функція частоти обміну повідомленнями

З рисунка 3.10 також можна помітити, що зміна розміру процесів програми при одночасному зміні швидкості передачі повідомлень мало впливає на накладні витрати на безвідмовність, криві для різних розмірів купи процесу програми майже перекриваються.

На рисунку 3.11 (а) показано накладні витрати на контрольні точки програми для трьох розмірів зображень у залежності від частоти їх встановлення. Для трьох варіантів програми накладні витрати зростають зі зменшенням інтервалу контрольних точок, причина полягає в тому, що з меншими інтервалами (вищою частотою контрольних точок) виконується більше контрольних точок. При використанні політики неблокуючого встановлення контрольних точок процедури контрольних точок *фактично* виконуються розгалуженим потоком, тоді як основна програма одночасно продовжує виконання. Однак, певний час виконання все ще втрачається, коли ядро ОС перемикає контекст і замінює один із процесів. Це незначне збільшення накладних витрат зростає з великими розмірами зображень (більше кілобайт для запису на диск), а отже, і з відхиленням накладних витрат трьох кривих на рисунку 3.11 (а) як інтервал створення контрольних точок зменшується.

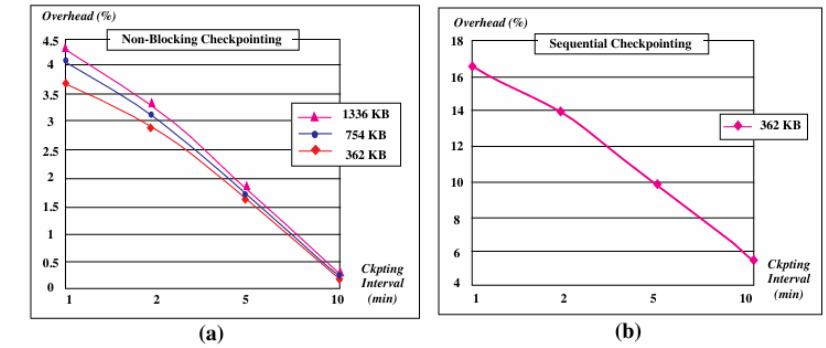

Рисунок 3.11 - Накладні витрати на безвідмовність як функція інтервалу контрольної точки

За аналогічних умов швидкості передачі повідомлень та розміру зображення, реєстрація повідомлень *з алгоритмом координованого встановлення контрольних точок* [8] показала менші накладні витрати для вищих частот контрольних точок *(1,8%* для інтервалу 2 хвилини), але наш алгоритм працює краще зі зменшенням частоти контрольних точок (0,2% накладних витрат порівняно з *0,3%* для інтервалу 10 хвилин). Це незважаючи на додаткову складність нашого алгоритму, спричинену врахуванням збоїв, що виникають, поки міжпроцесні повідомлення перебувають у тимчасовому стані. Рисунок 3.11 (b) підкреслює перевагу неблокуючої контрольної точки. Для найпомірніших накладних витрат (362 розмір програм бази знань та інтервал контрольних точок 10 хв) накладні витрати на послідовну контрольну точку становили *5,72%* порівняно з *0,12%* для неблокуючої контрольної точки.

Це дослідження продуктивності показує, що надійний протокол розподілених обчислень FADI продемонстрував низькі накладні витрати навіть із завищеними швидкостями обміну повідомленнями, що доводить доцільність системи для розподілених програм з інтенсивним зв'язком. Однак, більші накладні витрати (хоча все ще прийнятні) були виміряні для коротших інтервалів контрольних точок. Завдяки прозорості реалізації нашого надійного алгоритму розподілених обчислень, частоту контрольних точок можна налаштувати без будь-яких змін у коді програми. Вимоги до програм, цільових для FADI (збільшений час виконання), дозволяють використовувати вищі частоти контрольних точок і, отже, матимуть низькі накладні витрати під час роботи під

керуванням FADI.

Ми припускаємо, що ці результати будуть дійсними і на сучасних апаратних платформах. Мережі Fast Ethernet можуть забезпечувати швидкість зв'язку до гігабіту на секунду, а остання станція Sun SPARCstation *«ULTRA 80»* працює на частоті 450 МГц і може вмістити 4 ГБ оперативної пам'яті.

3.4 Висновки по розділу

У третьому розділі було розглянуто практичну реалізацію та оцінювання ефективності fault-tolerant рішень у розподілених системах. Проаналізовано сучасні платформи та інструменти для побудови відмовостійких систем, а також архітектурні підходи до організації розподілених сервісів. Проведено оцінювання ефективності та продуктивності реалізованих рішень, що дозволило визначити рівень їх надійності, масштабованості та здатності забезпечувати безперервну роботу в умовах виникнення збоїв. Отримані результати підтверджують доцільність використання fault-tolerant підходів для підвищення стабільності та ефективності сучасних програмних систем у розподілених середовищах.

ВИСНОВКИ

У ході виконання бакалаврської роботи було досліджено підходи та методи розробки fault-tolerant систем у розподілених середовищах, що є комплексним процесом, спрямованим на створення надійних, масштабованих і стійких до відмов програмних рішень. У роботі проаналізовано існуючі архітектурні підходи до побудови розподілених систем, розглянуто причини виникнення збоїв, а також досліджено сучасні технології та інструменти забезпечення відмовостійкості. Процес дослідження охопив аналіз теоретичних основ, моделювання архітектурних рішень, вибір відповідних технологій і оцінку ефективності їх застосування, що дозволило сформувати цілісне бачення побудови fault-tolerant систем.

Ми розпочали з аналізу вимог до розподілених систем, визначивши ключові функціональні та нефункціональні характеристики, зокрема надійність, доступність, масштабованість і узгодженість даних. Було досліджено базові принципи проєктування, такі як надлишковість, ізоляція компонентів, автоматичне відновлення та балансування навантаження. Моделювання архітектури дозволило структурувати взаємодію між компонентами системи, визначити критичні точки відмов і сформувати підхід до їх мінімізації. Використання діаграм (Use Case, Sequence, компонентних і розгортання) сприяло кращому розумінню логіки роботи системи та стало основою для подальшого аналізу.

У процесі дослідження було проаналізовано причини виникнення відмов у розподілених середовищах, включаючи апаратні, програмні та мережеві збої. Розглянуто методи виявлення та діагностики помилок, такі як моніторинг, логування та трасування. Значну увагу приділено механізмам обробки збоїв, серед яких повторні спроби виконання операцій, реплікація даних, резервування ресурсів, а також використання шаблонів проєктування, зокрема Circuit Breaker, Retry і Bulkhead. Це дозволило сформувати підхід до побудови систем, здатних зберігати працездатність навіть у випадку часткових відмов компонентів.

Практична частина роботи охопила дослідження платформних технологій і багаторівневих архітектур для реалізації fault-tolerant систем. Було розглянуто підходи до організації взаємодії між сервісами, зокрема використання мікросервісної архітектури та асинхронного обміну повідомленнями. Оцінка ефективності запропонованих рішень показала, що впровадження механізмів відмовостійкості значно підвищує стабільність системи, зменшує ризик простоїв і забезпечує безперервність надання сервісів користувачам навіть у разі виникнення збоїв.

Загалом, результати роботи підтверджують, що розробка fault-tolerant систем у розподілених середовищах є критично важливою для сучасних програмних рішень. Перевагами запропонованих підходів є підвищення надійності, гнучкість масштабування та здатність системи адаптуватися до змін навантаження. Водночас впровадження таких рішень супроводжується ускладненням архітектури та збільшенням витрат на підтримку інфраструктури. У майбутньому доцільним є подальший розвиток автоматизованих механізмів відновлення, вдосконалення систем моніторингу та застосування інтелектуальних методів аналізу збоїв, що дозволить підвищити ефективність і надійність розподілених систем.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Tanenbaum, A. S., & Van Steen, M. (2017). *Distributed Systems: Principles and Paradigms*. Pearson.
2. Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). *Distributed Systems: Concepts and Design*. Addison-Wesley.
3. Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media.
4. Newman, S. (2021). *Building Microservices*. O'Reilly Media.
5. Richardson, C. (2018). *Microservices Patterns*. Manning Publications.
6. Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. https://martinfowler.com/eaaCatalog/
7. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns*. Pearson.
8. Microsoft. (2025). Azure Architecture Center. https://learn.microsoft.com/en-us/azure/architecture/
9. Microsoft. (2025). Reliability in Azure Applications. https://learn.microsoft.com/en-us/azure/architecture/framework/resiliency/
10. AWS. (2025). AWS Well-Architected Framework. https://docs.aws.amazon.com/wellarchitected/
11. Google Cloud. (2025). Reliability Engineering. https://cloud.google.com/architecture/framework/reliability
12. Nygard, M. (2018). *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf.
13. Basiri, A., et al. (2016). Chaos Engineering. IEEE Software, 33(3), 35–41.
14. Netflix. (2025). Chaos Monkey. https://netflix.github.io/chaosmonkey/
15. Burns, B., & Beda, J. (2019). *Kubernetes: Up and Running*. O'Reilly Media.
16. Kubernetes Documentation. (2025). https://kubernetes.io/docs/
17. Docker Documentation. (2025). https://docs.docker.com/

18. Apache Kafka Documentation. (2025). https://kafka.apache.org/documentation/

19. RabbitMQ Documentation. (2025). https://www.rabbitmq.com/documentation.html

20. Resilience4j Documentation. (2025). https://resilience4j.readme.io/docs

21. Hystrix. (2024). Netflix OSS. https://github.com/Netflix/Hystrix

22. OWASP. (2024). Top Ten Security Risks. https://owasp.org/www-project-top-ten/

23. Chen, J., & Li, X. (2024). Fault-tolerant cloud systems. IEEE Transactions on Cloud Computing, 12(2), 112–125.

24. Kumar, S., & Sharma, R. (2023). Microservices fault tolerance strategies. Journal of Software Engineering, 11(2), 67–82.

25. Zhang, Q., Chen, M., & Li, L. (2025). Distributed system reliability. ACM Computing Surveys, 57(1), 1–34.

26. Brewer, E. (2012). CAP Theorem Revisited. IEEE Computer, 45(2), 23–29.

27. Vogels, W. (2009). Eventually Consistent Systems. ACM Queue, 6(6), 14–19.

28. Lamport, L. (1998). The Part-Time Parliament. ACM Transactions on Computer Systems.

29. Ongaro, D., & Ousterhout, J. (2014). In Search of an Understandable Consensus Algorithm (Raft).

30. Gray, J., & Reuter, A. (1992). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

31. Microsoft. (2025). .NET Microservices Architecture. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

32. Azure Service Fabric Documentation. (2025). https://learn.microsoft.com/en-us/azure/service-fabric/

33. Prometheus Monitoring. (2025). https://prometheus.io/docs/

34. Grafana Documentation. (2025). https://grafana.com/docs/

35. ELK Stack Documentation. (2025). https://www.elastic.co/what-is/elk-stack

36. OpenTelemetry. (2025). https://opentelemetry.io/docs/

37. Google SRE Book. (2023). https://sre.google/books/

38. Bass, L., Clements, P., & Kazman, R. (2013). *Software Architecture in Practice*. Addison-Wesley.

39. Sommerville, I. (2016). *Software Engineering*. Pearson.

40. ISO/IEC 25010. (2011). Systems and Software Quality Requirements and Evaluation (SQuaRE).

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: "Розробка fault-tolerant систем у розподілених середовищах"

Обсяг пояснювальної записки: 85 аркушів

Дата закінчення бакалаврської роботи 10 червня 2026р.

Підпис студента _____