

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО–КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Розробка сервісу з моделювання прискорення
процесу обчислювання великих даних за
технологією orleans»**

Виконав студент групи ІПЗм–24–1
Гордієнко Микола Геннадійович

Керівник роботи:
ВЛАСЕНКО Олег Васильович

Рецензент:
ЄФРЕМОВ Юрій Миколайович

«ЗАТВЕРДЖУЮ»

зав. кафедри інженерії

програмного забезпечення

Тетяна ВАКАЛЮК

«01» жовтня 2024 р.

ЗАВДАННЯ

на кваліфікаційну роботу

Здобувач вищої освіти: **ГОРДІЄНКО Микола Геннадійович**

Керівник роботи: **ВЛАСЕНКО Олег Васильович**

Тема роботи: **«Розробка сервісу з моделювання прискорення процесу обчислювання великих даних за технологією orleans»**, затверджена Наказом закладу вищої освіти від «18» вересня 2025 р. № 491/с.

Вихідні дані для роботи: Об'єкт дослідження: процеси швидкісної обробки даних у високонавантажених розподілених системах. **Предмет дослідження:** технологія Orleans та модель віртуальних акторів для забезпечення консистентності даних без застосування блокувань.

Консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Власенко О.В.	22.09.2025р.	22.09.2025р.
2	Власенко О.В.	22.09.2025р.	02.10.2025р.
3	Власенко О.В.	22.09.2025р.	12.11.2025р.

РЕФЕРАТ

Кваліфікаційна робота освітнього ступеня «магістр» присвячена дослідженню ефективності моделі віртуальних акторів для керування станом у високонавантажених системах.

Пояснювальна записка складається зі вступу, трьох розділів, висновків та списку використаних джерел. Робота містить 108 сторінок, 11 ілюстрацій, 10 таблиць, 4 додатки та 40 посилань на зовнішні джерела.

Метою роботи є розробка сервісу для моделювання прискорення процесу обчислювання великих даних за технологією Orleans та порівняння з традиційним підходом на основі розподілених блокувань.

В роботі розроблено демонстраційну систему торгового стакану біржі з двома архітектурними реалізаціями: stateless (Redis + RedLock) та Orleans (Grains). Проведено експериментальне дослідження за трьома сценаріями навантаження. Встановлено, що Orleans забезпечує у 4–7 разів вищу пропускну здатність та у 10–11 разів нижчу латентність при високій конкуренції за ресурси.

КЛЮЧОВІ СЛОВА: ВІРТУАЛЬНІ АКТОРИ, ORLEANS, LOCK CONTENTION, ТОРГОВИЙ СТАКАН, РОЗПОДІЛЕНІ СИСТЕМИ.

ABSTRACT

The master's thesis is devoted to the study of the virtual actor model efficiency for state management in high-load systems.

The explanatory note consists of an introduction, three chapters, conclusions and a list of references. The work contains 108 pages, 11 illustrations, 10 tables, 4 appendices and 40 references to external sources.

The purpose of the work is to develop a service for modeling the acceleration of big data processing using Orleans technology and comparison with the traditional approach based on distributed locks.

A demonstration stock exchange order book system with two architectural implementations was developed: stateless (Redis + RedLock) and Orleans (Grains). Experimental research was conducted across three load scenarios. It was established that Orleans provides 4–7 times higher throughput and 10–11 times lower latency under high resource contention.

KEYWORDS: VIRTUAL ACTORS, ORLEANS, LOCK CONTENTION, ORDER BOOK, DISTRIBUTED SYSTEMS.

ЗМІСТ

ВСТУП	6
РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ПРИСКОРЕННЯ ОБЧИСЛЮВАЛЬНИХ ПРОЦЕСІВ ЗА ТЕХНОЛОГІЄЮ ORLEANS	8
1.1 Аналіз існуючих підходів до керування станом у розподілених системах. ...	8
1.2 Технологія Orleans як рішення проблеми конкурентного доступу	11
1.3 Постановка задачі дослідження.....	15
Висновки до першого розділу.....	17
РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ДЕМОНСТРАЦІЙНОЇ СИСТЕМИ	17
2.1 Формальна модель торгового стакану	19
2.2 Алгоритм співставлення ордерів	23
2.3 Архітектура демонстраційної системи	27
2.4 Порівняльна реалізація архітектурних підходів	31
2.5 Методика експериментального дослідження.....	31
Висновки до другого розділу	40
РОЗДІЛ 3. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ	40
3.1 Вимоги до системи та середовище виконання.....	41
3.2 Інтерфейс користувача та засоби візуалізації	44
3.3 Проведення експериментів та аналіз результатів	49
3.4 Аналіз причин різниці у продуктивності.....	52
3.5 Рекомендації та межі застосування	56
Висновки до третього розділу.....	59
ВИСНОВКИ	60
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	61
ДОДАТКИ	64

ВСТУП

Актуальність теми. Проблема керування станом у високонавантажених системах є досить поширеною. Фінансові платформи, ігрові сервери, системи бронювання – всі вони мають справу з тисячами запитів на секунду до спільних ресурсів.

Візьмемо до прикладу торговий стакан біржі. Сотні трейдерів одночасно намагаються купити чи продати акції за однаковою ціною. Система повинна обробити кожен запит, не втратити жодного ордера і при цьому працювати швидко.

Традиційний підхід – використання розподілених блокувань [33]. Перед зміною даних процес захоплює блокування, робить свою справу і звільняє його. Проблема в тому, що при великому навантаженні процеси більше чекають у черзі на блокування, ніж працюють. У наших експериментах цей час очікування сягав 65% [1, 2].

Інший варіант – модель віртуальних акторів. Кожен актор обробляє запити послідовно в одному потоці. Блокування не потрібні – консистентність гарантована самою архітектурою. Цей підхід реалізовано у фреймворку Microsoft Orleans [29].

Але чи справді Orleans швидший? І наскільки? На ці питання важко відповісти без практичного порівняння. Існуючі дослідження здебільшого теоретичні або використовують синтетичні тести.

Мета роботи – розробка сервісу для порівняння ефективності традиційної архітектури та Orleans на прикладі торгового стакану.

Для досягнення мети потрібно вирішити такі завдання:

1. Дослідити причини деградації продуктивності при використанні розподілених блокувань.
2. Вивчити модель віртуальних акторів Orleans.
3. Розробити демонстраційну систему торгового стакану з двома реалізаціями.

4. Провести експерименти та порівняти результати.

Об'єктом дослідження є процеси керування станом у розподілених системах з високою конкуренцією за ресурси.

Предметом дослідження є модель віртуальних акторів як альтернатива розподіленим блокуванням.

Наукова новизна полягає у проведенні практичного порівняння двох архітектур на реалістичному сценарії торгового стакану. Визначено поріг ефективності Orleans – понад 100 одночасних запитів на ресурс. Встановлено головну причину деградації традиційної архітектури – 65% часу на очікування блокувань.

Практичне значення полягає у створенні методики порівняння архітектур та рекомендацій щодо вибору підходу залежно від характеристик навантаження.

Апробація результатів роботи. Матеріали кваліфікаційної роботи доповідалися на «Impact of Artificial Intelligence and Other Technologies on Sustainable Development» – 11 – 12 грудня, що присвячена дослідженням та інноваційним рішенням, що впливають на сучасний світ. Посилання на публікації – Додаток Д.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ПРИСКОРЕННЯ ОБЧИСЛЮВАЛЬНИХ ПРОЦЕСІВ ЗА ТЕХНОЛОГІЄЮ ORLEANS

1.1 Аналіз існуючих підходів до керування станом у розподілених системах

Сучасні високонавантажені системи – фінансові платформи [22], системи електронної комерції, ігрові сервери – характеризуються інтенсивним конкурентним доступом до спільних ресурсів. Типовим прикладом є торговий стакан біржової платформи, який агрегує заявки на купівлю та продаж фінансових інструментів. Такі системи отримують тисячі оновлень на секунду, мають жорсткі вимоги до латентності та повинні гарантувати консистентність стану при одночасному доступі множини клієнтів. Ці характеристики роблять подібні системи показовим об'єктом для дослідження проблем керування станом.

Традиційний підхід до побудови високонавантажених веб–додатків базується на архітектурі без збереження стану [18] на сервері. У такій архітектурі кожен запит обробляється незалежно, а весь стан зберігається у зовнішньому сховищі даних. Цей підхід має очевидні переваги для горизонтального масштабування: будь–який сервер може обробити будь–який запит, оскільки не залежить від локального стану. Проте при роботі зі спільними ресурсами виникають фундаментальні проблеми забезпечення консистентності даних.

Типова архітектура системи без збереження стану включає кілька компонентів. Балансувальник навантаження розподіляє вхідні запити між серверами додатків. Кожен сервер додатків є ідентичною копією, яка містить бізнес–логіку обробки запитів. Зовнішнє сховище, зазвичай Redis або інша розподілена база даних, зберігає актуальний стан системи. Всі сервери звертаються до спільного сховища для читання та модифікації стану.

Проблема виникає, коли кілька серверів одночасно намагаються модифікувати один і той самий ресурс. Якщо два сервери паралельно читають стан, модифікують його та записують назад, виникає класична проблема гонки даних [20]. Обидва сервери прочитають однаковий початковий стан, кожен внесе

свої зміни, і при записі результат одного буде втрачено, оскільки друга операція запису перезапише результат першої.

Стандартним рішенням проблеми гонки даних є використання механізму розподілених блокувань. Перед модифікацією ресурсу процес повинен отримати ексклюзивне блокування на цей ресурс. Після завершення операції блокування звільняється. Протокол RedLock, розроблений автором Redis [33], є найпоширенішою реалізацією розподілених блокувань для систем на базі Redis.

Механізм розподілених блокувань вирішує проблему консистентності, але створює нові проблеми при високому навантаженні. Перша проблема полягає у конкуренції за блокування. Коли множина процесів одночасно намагається отримати блокування на один ресурс, лише один з них успішно його отримує. Всі інші процеси змушені очікувати звільнення блокування. При високій інтенсивності запитів час очікування зростає нелінійно, оскільки черга очікуючих процесів постійно збільшується.

Дослідження показують, що при навантаженні понад тисячу одночасних запитів до одного ресурсу процеси можуть витрачати до шістдесяти п'яти відсотків часу на очікування блокувань [1] замість виконання корисної роботи. Це призводить до катастрофічної деградації продуктивності системи. Пропускна здатність не зростає пропорційно до збільшення кількості серверів, оскільки вузьким місцем стає доступ до спільного ресурсу, а не обчислювальна потужність.

Друга проблема пов'язана з таймаутами блокувань. Кожне блокування має обмежений час життя для запобігання ситуації, коли процес отримав блокування і завершився аварійно без його звільнення. Якщо операція не встигає завершитися до закінчення таймауту, блокування автоматично звільняється, і інший процес може його отримати. Це створює ризик одночасної модифікації ресурсу двома процесами, що порушує гарантії консистентності.

Третя проблема полягає у ризику взаємних блокувань. Якщо операція потребує одночасного блокування кількох ресурсів, може виникнути ситуація, коли процес А заблокував ресурс X і очікує ресурс Y, тоді як процес B заблокував

ресурс Y і очікує ресурс X. Жоден з процесів не може продовжити виконання, і система зависає.

При використанні розподілених блокувань обробка кожного запиту включає наступну послідовність операцій: запит на отримання блокування, очікування якщо блокування зайняте, читання поточного стану зі сховища, обробка бізнес-логіки, запис оновленого стану, звільнення блокування. Кожна з цих операцій додає затримку. Мережева комунікація для отримання блокування займає одиниці мілісекунд. Очікування звільнення блокування може тривати десятки або сотні мілісекунд при високому навантаженні.

Експериментальне тестування типової архітектури з Redis та RedLock показує характерну картину деградації. При навантаженні до ста запитів на секунду система працює стабільно з медіанною затримкою близько п'яти мілісекунд. При збільшенні навантаження до п'ятисот запитів на секунду медіанна затримка зростає до двадцяти мілісекунд, а дев'яносто дев'ятий перцентиль сягає ста мілісекунд. При тисячі запитів на секунду система починає втрачати запити через таймаути, а затримка для успішно оброблених запитів перевищує двісті мілісекунд.

У таблиці 1.1 наведено порівняння характеристик продуктивності традиційної архітектури при різних рівнях навантаження.

Таблиця 1.1

Характеристики продуктивності традиційної архітектури

Навантаження (запитів/с)	Медіанна затримка (мс)	P99 затримка (мс)	Втрата запитів (%)
100	5	15	0
500	20	100	0
1000	85	350	2–5
2000	200	800	5–10

Як видно з таблиці, при збільшенні навантаження вдесятеро пропускну здатність зростає лише вдвічі, тоді як затримка збільшується майже в сорок разів.

Це демонструє нелінійну деградацію продуктивності, характерну для систем з високою конкуренцією за блокування.

Складність коду є ще одним суттєвим недоліком традиційного підходу. Розробник повинен явно керувати життєвим циклом блокувань, обробляти таймаути та повторні спроби, забезпечувати коректне звільнення блокувань навіть при виникненні виключень, реалізовувати логіку виявлення та обробки взаємних блокувань. Аналіз типового коду показує, що близько шістдесяти відсотків коду припадає на інфраструктурну логіку керування блокуваннями та обробку помилок. Лише сорок відсотків коду безпосередньо реалізує бізнес-логіку.

Підсумовуючи, традиційна архітектура без збереження стану з використанням зовнішнього сховища та розподілених блокувань має фундаментальні обмеження для систем з високою конкуренцією за спільні ресурси. Проблеми конкуренції за блокування, таймаутів та взаємних блокувань призводять до нелінійної деградації продуктивності при зростанні навантаження. Ці обмеження обумовлюють необхідність пошуку альтернативних архітектурних підходів.

1.2 Технологія Orleans як рішення проблеми конкурентного доступу

Проблеми традиційних архітектур, описані у попередньому підрозділі, обумовлюють необхідність пошуку альтернативних підходів. Одним із таких підходів є модель віртуальних акторів, реалізована у фреймворку Microsoft Orleans [8]. Ця технологія пропонує принципово інший спосіб організації роботи зі спільним станом, який архітектурно усуває потребу у розподілених блокуваннях.

Ключова ідея Orleans полягає у тому, що замість винесення стану у зовнішнє сховище з подальшою конкуренцією за доступ до нього, стан інкапсулюється всередині спеціальних об'єктів – грейнів [29, 30]. Кожен грейн є ізольованою одиницею, яка зберігає свій стан в оперативній пам'яті та обробляє всі запити до цього стану послідовно.

Розглянемо, як це працює на прикладі торгового стакану. У традиційній архітектурі стан стакану для торгової пари BTC–USD зберігається у Redis, і кожен сервер при обробці ордера повинен отримати блокування, прочитати стан, модифікувати його та записати назад. В архітектурі Orleans створюється грейн `OrderBookGrain` з ідентифікатором "BTC–USD", який містить стан стакану у своїх полях та надає методи для роботи з ордерами.

Грейн складається з трьох компонентів. Ідентичність визначається типом грейна (`IOrderBookGrain`) та первинним ключем (наприклад, "BTC–USD"). Комбінація типу та ключа є унікальною у всій системі. Поведінка визначається інтерфейсом з методами: `PlaceLimitOrder`, `PlaceMarketOrder`, `CancelOrder`, `GetSnapshot`. Стан зберігається у приватних полях класу: колекції ордерів на купівлю та продаж, список виконаних угод, лічильник послідовності.

Фундаментальною гарантією Orleans є єдина активація [11]: у будь-який момент часу у всій системі існує щонайбільше один активний екземпляр грейна з заданим ідентифікатором. Це означає, що грейн `OrderBookGrain` для пари "BTC–USD" існує в одному екземплярі, навіть якщо кластер складається з десятків серверів. Усі запити на розміщення ордерів для цієї пари автоматично направляються до єдиного екземпляра грейна.

Середовище виконання Orleans організує чергу запитів до грейна та виконує їх послідовно, один за одним. Коли до грейна одночасно надходять сто ордерів від різних клієнтів, вони поміщаються у чергу та обробляються по черзі. Кожен метод грейна виконується повністю до початку наступного. Це автоматично забезпечує консистентність стану без жодних блокувань, семафорів чи інших примітивів синхронізації.

На рисунку 1.1 показано порівняння обробки конкурентних запитів у двох архітектурах.

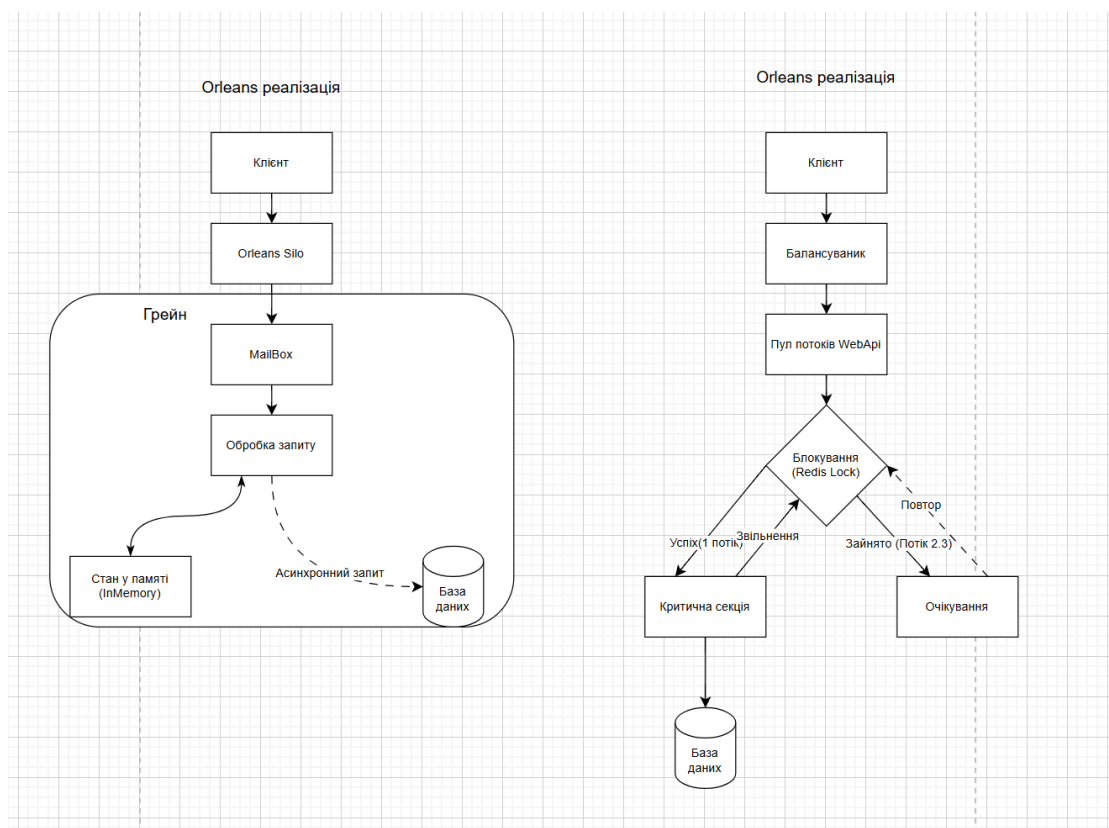


Рисунок 1.1 Порівняння обробки конкурентних запитів: традиційна архітектура vs Orleans

У традиційній архітектурі п'ять серверів конкурують за одне блокування Redis. Чотири з них очікують, поки один обробляє запит. В архітектурі Orleans всі запити направляються до одного грейна, який обробляє їх послідовно без очікування – черга запитів замість черги за блокуванням.

Життєвий цикл грейна керується автоматично. При першому зверненні до грейна `OrderBookGrain("BTC-USD")` система автоматично створює його екземпляр на одному із серверів кластера. Грейн залишається активним в оперативній пам'яті поки до нього надходять запити. Якщо торгова пара стає неактивною протягом певного часу, грейн автоматично деактивується для звільнення ресурсів. При наступному зверненні грейн буде активовано знову, можливо на іншому сервері кластера.

Сервери у кластері Orleans називаються силосами. Кожен силос може містити тисячі активних грейнів. Розподіл грейнів між силосами здійснюється автоматично на основі хешування ідентифікаторів. Клієнтський код не знає і не

потребує знати, на якому силосі знаходиться конкретний грейн – система автоматично маршрутизує запити.

Важливою перевагою є спосіб написання коду. Метод обробки ордера у грейні виглядає як звичайний послідовний код без будь-якої інфраструктурної логіки. Порівняємо підходи на прикладі розміщення лімітного ордера.

У традиційній архітектурі код містить: отримання блокування з обробкою таймауту, читання стану з Redis, десеріалізацію, виконання бізнес-логіки, серіалізацію, запис у Redis, звільнення блокування у блоці `finally`. Близько шістдесяти відсотків коду – інфраструктура.

В Orleans код містить лише бізнес-логіку: створення ордера, виклик алгоритму співставлення, оновлення внутрішніх колекцій, повернення результату. Блокування, серіалізація, мережева комунікація – все це відсутнє, бо стан вже в пам'яті грейна, а послідовність гарантується архітектурою.

У таблиці 1.2 наведено порівняння характеристик двох підходів.

Таблиця 1.2

Порівняння традиційної архітектури та Orleans

Характеристика	Традиційна (Redis + Locks)	Orleans (Grains)
Розташування стану	Зовнішнє сховище	Пам'ять грейна
Механізм консистентності	Розподілені блокування	Єдина активація
Затримка доступу до стану	Мережева (мс)	Локальна (мкс)
Код синхронізації	Явний, ~60% коду	Відсутній
Масштабування	Вузьке місце на блокуваннях	Розподіл грейнів по силосах
Ризик втрати даних	Таймаути блокувань	Відсутній

Серед існуючих реалізацій моделі віртуальних акторів Orleans є найбільш зрілою для платформи .NET. Фреймворк розроблений Microsoft Research та використовується у промислових системах з мільйонами користувачів: Xbox

Live для керування ігровими сесіями, Azure PlayFab для ігрової аналітики, Halo [32] для мультиплеєрних матчів. Ці впровадження підтверджують здатність технології витримувати екстремальні навантаження.

Для даного дослідження обрано Orleans як технологію реалізації системи на основі віртуальних акторів. Вибір обумовлений вбудованою гарантією єдиної активації, інтеграцією з екосистемою .NET та ASP.NET Core, наявністю детальної документації та прикладів, підтвердженою ефективністю у промислових системах.

Для забезпечення об'єктивного порівняння реалізовано дві версії демонстраційної системи торгового стакану: традиційну на базі Redis з блокуваннями RedLock та версію на Orleans. Обидві версії реалізують ідентичну бізнес-логіку та надають єдиний API, що дозволяє провести коректне порівняння продуктивності.

1.3 Постановка задачі дослідження

Проведений аналіз проблем керування станом у високонавантажених системах та альтернативного підходу на основі моделі віртуальних акторів дозволяє сформулювати задачу дослідження.

Метою дослідження є розробка сервісу з моделювання прискорення процесу обчислювання великих даних за технологією Orleans та експериментальна оцінка ефективності цього підходу порівняно з традиційною архітектурою на основі розподілених блокувань.

Для досягнення поставленої мети необхідно вирішити наступні завдання. Перше завдання полягає у проектуванні архітектури демонстраційної системи торгового стакану, яка дозволить реалізувати обидва підходи з ідентичною бізнес-логікою для забезпечення об'єктивного порівняння. Друге завдання передбачає реалізацію традиційної версії системи з використанням зовнішнього сховища стану та механізму розподілених блокувань для забезпечення консистентності даних. Третє завдання полягає у реалізації версії системи на основі моделі віртуальних акторів з використанням грейнів для зберігання стану торгового стакану. Четверте завдання передбачає розробку методології

експериментального дослідження з набором тестових сценаріїв, які моделюють різні умови навантаження. П'яте завдання полягає у проведенні серії експериментів та зборі метрик продуктивності для обох реалізацій. Шосте завдання передбачає виконання порівняльного аналізу отриманих результатів та формулювання рекомендацій щодо застосування кожного підходу.

Для об'єктивного порівняння двох архітектурних підходів визначено наступні критерії оцінки ефективності. Пропускна здатність визначається як кількість оброблених ордерів за одиницю часу. Цей показник характеризує максимальну продуктивність системи та є критичним для торгових платформ з високою інтенсивністю операцій. Латентність визначається як час від надходження ордера до завершення його обробки. Для оцінки латентності використовуються перцентилі розподілу: медіана, дев'яносто п'ятий перцентиль та дев'яносто дев'ятий перцентиль. Використання перцентилів дозволяє оцінити не лише типовий час відгуку, але й поведінку системи у найгірших випадках. Надійність обробки визначається як відсоток успішно оброблених ордерів від загальної кількості надісланих. Цей показник характеризує здатність системи обробляти всі запити без втрат навіть при високому навантаженні. Час очікування блокування є специфічним показником для традиційної архітектури та визначає частку часу, яку процеси витрачають на очікування доступу до заблокованих ресурсів. Складність реалізації оцінюється за кількістю рядків коду, необхідних для реалізації еквівалентної функціональності в обох підходах.

Експериментальне дослідження передбачає проведення трьох типів тестових сценаріїв. Перший сценарій моделює ситуацію максимальної конкуренції, коли велика кількість ордерів одночасно надходить на один ціновий рівень. Цей сценарій дозволяє оцінити поведінку системи при найвищому рівні конкуренції за спільний ресурс. Другий сценарій моделює ситуацію різкого обвалу ринку, коли великий ринковий ордер послідовно виконується проти множини менших ордерів у стакані. Цей сценарій перевіряє коректність алгоритму співставлення та атомарність операцій. Третій сценарій моделює тривале високочастотне навантаження з ордерами на різних цінових рівнях. Цей

сценарій дозволяє оцінити стабільність продуктивності системи протягом тривалого періоду роботи.

Об'єктом дослідження є процес обчислювання великих даних у розподілених системах з інтенсивним конкурентним доступом до спільних ресурсів. Предметом дослідження є технологія Microsoft Orleans та методи моделювання обчислювальних процесів на основі віртуальних акторів як засіб прискорення обробки даних.

Дослідження обмежується порівнянням двох конкретних реалізацій: традиційної архітектури з використанням Redis як сховища стану та механізму розподілених блокувань, та архітектури на основі віртуальних акторів з використанням фреймворку Orleans. Обидві реалізації виконуються на ідентичному апаратному забезпеченні для забезпечення коректності порівняння.

Результатом дослідження мають стати кількісні характеристики ефективності обох підходів та обґрунтовані рекомендації щодо вибору архітектури залежно від характеристик навантаження системи.

Висновки до першого розділ

У розділі проаналізовано проблему керування станом у високонавантажених системах.

Торговий стакан обрано як предметну область через високу конкуренцію за спільні ресурси. Тисячі клієнтів одночасно змінюють ті самі цінові рівні. Це створює ідеальні умови для порівняння архітектурних підходів.

Традиційний підхід – stateless-сервери з Redis та розподіленими блокуваннями – має суттєві обмеження. При навантаженні понад 1000 операцій на секунду до 65% часу витрачається на очікування блокувань. Латентність зростає нелінійно. З'являються втрати даних через таймаути.

Модель віртуальних акторів вирішує проблему на архітектурному рівні. Кожен актор обробляє запити в одному потоці. Блокування не потрібні – консистентність гарантована самою моделлю.

Для реалізації обрано Microsoft Orleans. Це production-ready фреймворк для .NET, що використовується в Xbox Live, Azure PlayFab та інших системах з мільйонами користувачів.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ДЕМОНСТРАЦІЙНОЇ СИСТЕМИ

2.1 Формальна модель торгового стакану

Для забезпечення коректності реалізації та можливості верифікації результатів експериментів необхідно побудувати математичну модель торгового стакану. Формалізація визначає структуру даних, допустимі операції та інваріанти консистентності, які система повинна підтримувати незалежно від обраної архітектури. Наявність формальної моделі дозволяє перевірити, що обидві реалізації – традиційна та на основі віртуальних акторів – семантично еквівалентні та відрізняються лише нефункціональними характеристиками.

Торговий стакан є центральною структурою даних біржової системи, яка агрегує активні заявки на купівлю та продаж певного фінансового інструменту. Формально стакан для однієї торгової пари визначається як кортеж $OB = (S, Bids, Asks, Trades, seq)$, де S є символом торгової пари, наприклад "BTC–USD". Компонент $Bids$ представляє впорядковану множину ордерів на купівлю, а $Asks$ – впорядковану множину ордерів на продаж. Компонент $Trades$ зберігає послідовність виконаних угод, що виникли внаслідок співставлення ордерів. Лічильник seq використовується для генерації унікальних послідовних номерів, які визначають порядок надходження ордерів та є критичними для справедливого розподілу пріоритетів.

Ордер є базовою одиницею взаємодії трейдера з біржею та представляє намір купити або продати певний обсяг активу за вказаною ціною. Формально ордер визначається як кортеж $Order = (id, side, type, price, qty, filled, status, seqNum, ts)$. Ідентифікатор id є унікальним у межах всієї системи та генерується як UUID при створенні ордера. Поле $side$ приймає значення Buy для ордерів на купівлю або $Sell$ для ордерів на продаж. Тип ордера $type$ визначає механізм виконання: лімітні ордери ($Limit$) розміщуються у стакані та очікують зустрічної заявки за вказаною ціною, тоді як ринкові ордери ($Market$) виконуються негайно за найкращою доступною ціною. Поле $price$ містить бажану ціну для лімітних

ордерів або залишається порожнім для ринкових. Початковий обсяг `qty` фіксується при створенні ордера, а поле `filled` відстежує вже виконану кількість, яка зростає при кожному частковому співставленні.

Статус ордера відображає його поточний стан у життєвому циклі та може набувати одного з п'яти значень. Новостворений ордер отримує статус `Pending` до моменту його обробки системою. Після додавання до стакану статус змінюється на `Active`, що означає готовність до співставлення. При частковому виконанні, коли виконано частину обсягу але не весь, ордер переходить у стан `PartiallyFilled`. Повне виконання всього обсягу переводить ордер у термінальний стан `Filled`. Альтернативним термінальним станом є `Cancelled`, який встановлюється при скасуванні ордера користувачем або системою.

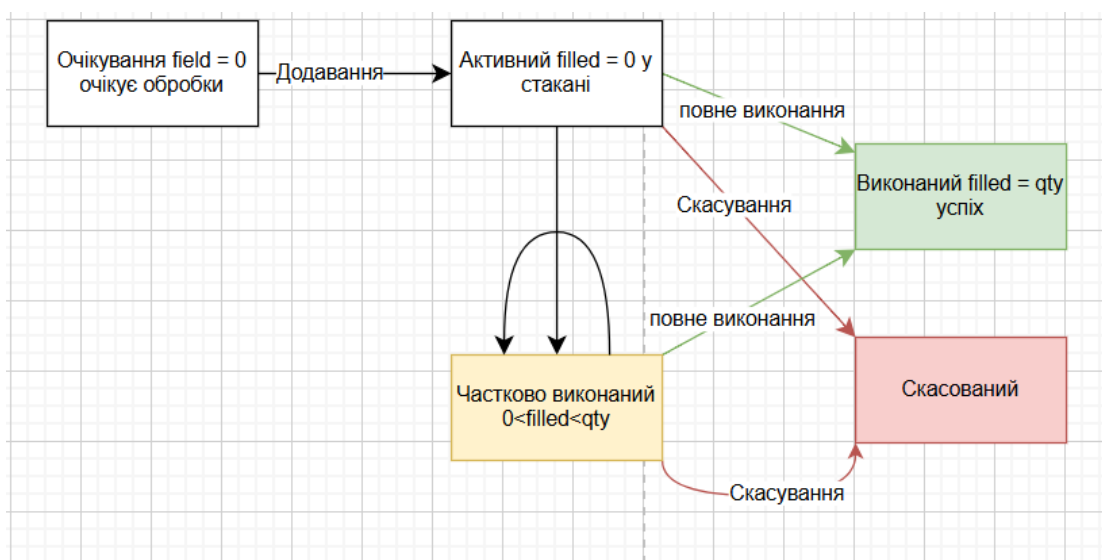


Рис. 2.1. Діаграма станів життєвого циклу ордера

Для роботи з ордерами визначаються допоміжні функції, які спрощують опис алгоритмів. Функція $\text{remaining}(o) = o.qty - o.filled$ обчислює залишковий обсяг ордера, доступний для виконання. Предикат $\text{isActive}(o)$ повертає істину, якщо ордер знаходиться у стані `Active` або `PartiallyFilled`, тобто може брати участь у співставленні. Предикат $\text{isFullyFilled}(o)$ перевіряє, чи повністю виконано ордер, що еквівалентно умові $o.filled = o.qty$.

Впорядкування ордерів у стакані реалізує принцип `price-time priority` [14], який є стандартом для біржових систем та забезпечує справедливий розподіл виконання. Суть принципу полягає у тому, що пріоритет надається ордерам з

найкращою ціною, а серед ордерів з однаковою ціною – тим, що надійшли раніше. Для множини Bids найкращою вважається найвища ціна, оскільки покупець, готовий платити більше, має отримати пріоритет. Формально порядок визначається як $o_1 \leq_{\text{Bids}} o_2$ тоді і тільки тоді, коли ціна o_1 більша за ціну o_2 , або при рівних цінах послідовний номер o_1 менший за номер o_2 . Для множини Asks найкращою є найнижча ціна, оскільки продавець, готовий продати дешевше, має пріоритет. Відповідно порядок визначається як $o_1 \leq_{\text{Asks}} o_2$ тоді і тільки тоді, коли ціна o_1 менша за ціну o_2 , або при рівних цінах послідовний номер o_1 менший.

Угода є результатом успішного співставлення двох ордерів і фіксує факт обміну активами між покупцем та продавцем. Формально угода визначається як кортеж $\text{Trade} = (\text{id}, \text{buyOrderId}, \text{sellOrderId}, \text{price}, \text{qty}, \text{ts})$, де id є унікальним ідентифікатором угоди, buyOrderId та sellOrderId є посиланнями на ордери учасників, price є ціною виконання, qty є обсягом угоди, а ts фіксує момент виконання. Ціна угоди визначається ціною пасивного ордера, тобто того, що вже знаходився у стакані на момент надходження зустрічної заявки. Це стандартне правило *maker-taker pricing*, яке стимулює надання ліквідності.

Консистентність стану торгового стакану визначається набором інваріантів, які повинні виконуватись після завершення кожної операції. Порушення будь-якого інваріанта свідчить про помилку в реалізації та може призвести до некоректної роботи системи або фінансових втрат учасників торгів.

Перший інваріант I_1 гарантує розділення сторін та формулюється наступним чином: для будь-якого ордера o , що належить множині Bids, поле side дорівнює Buy, і аналогічно для будь-якого ордера з множини Asks поле side дорівнює Sell. Цей інваріант є тривіальним, але його порушення призвело б до хаотичного співставлення ордерів однієї сторони між собою.

Другий інваріант I_2 є ключовим для коректності торгової системи та гарантує відсутність арбітражної можливості у стакані. Нехай bestBid позначає найвищу ціну серед активних ордерів на купівлю, а bestAsk – найнижчу ціну серед активних ордерів на продаж. Інваріант вимагає, щоб при непорожніх множинах Bids та Asks виконувалась нерівність $\text{bestBid} < \text{bestAsk}$. Іншими

словами, найкраща ціна купівлі завжди менша за найкращу ціну продажу. Якщо ця умова порушується ($\text{bestBid} \geq \text{bestAsk}$), це означає, що існують ордери, які можуть бути співставлені, і система повинна негайно виконати матчинг. Після завершення співставлення інваріант відновлюється.

Третій інваріант I_3 обмежує допустимі значення виконаного обсягу: для будь-якого ордера o виконується подвійна нерівність $0 \leq o.\text{filled} \leq o.\text{qty}$. Нижня межа гарантує, що виконаний обсяг не може бути від'ємним, верхня – що не можна виконати більше, ніж було замовлено.

Четвертий інваріант I_4 забезпечує консистентність між числовими полями ордера та його статусом. Ордер з нульовим виконаним обсягом може мати статус Pending, Active або Cancelled. Ордер з частково виконаним обсягом ($0 < \text{filled} < \text{qty}$) повинен мати статус PartiallyFilled. Повністю виконаний ордер ($\text{filled} = \text{qty}$) повинен мати статус Filled. Цей інваріант гарантує, що статус завжди коректно відображає фактичний стан виконання.

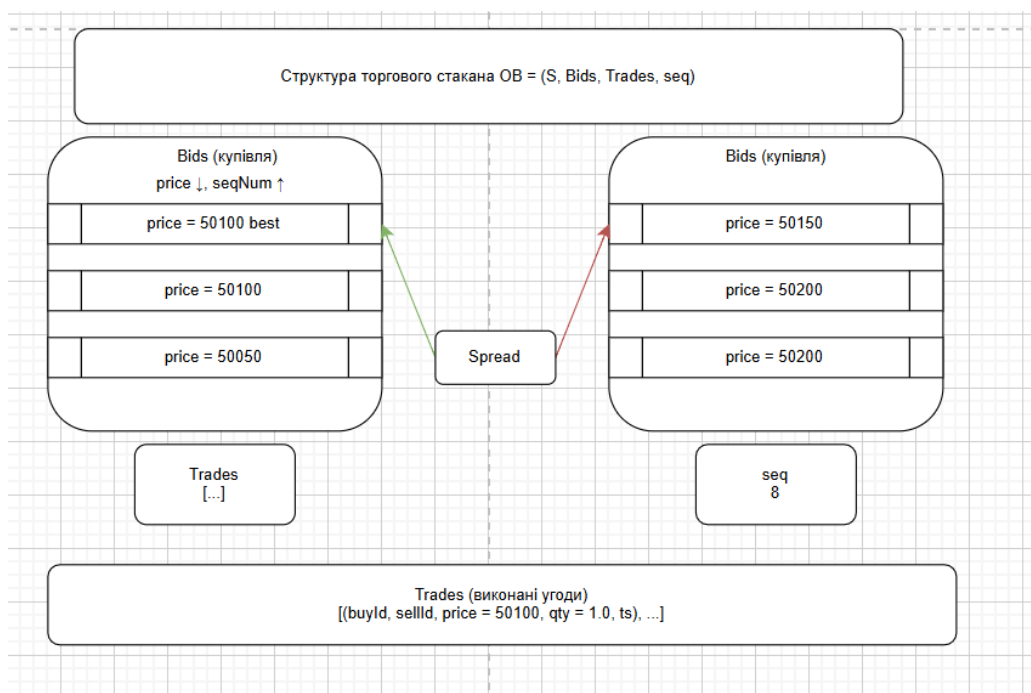


Рис. 2.2. Структура торгового стакану та інваріанти консистентності

Операції над стаканом формалізуються як функції, що трансформують стан системи. Операція додавання лімітного ордера $\text{AddLimit}: \text{OB} \times \text{Order} \rightarrow \text{OB} \times \text{Trade}^*$ приймає поточний стан стакану та новий ордер, виконує можливе співставлення, та повертає оновлений стан разом з множиною створених угод,

яка може бути порожньою. Аналогічно визначаються операції AddMarket для ринкових ордерів та Cancel для скасування. Ключовою властивістю є атомарність: кожна операція переводить стакан з одного консистентного стану в інший, і проміжні стани не спостерігаються ззовні.

Умова співставлення визначає, коли два ордери можуть утворити угоду. Для лімітного ордера на купівлю o_buy та найкращого ордера на продаж o_ask співставлення можливе, якщо ціна покупця не менша за ціну продавця ($o_buy.price \geq o_ask.price$) та ордер продавця є активним. Для ринкового ордера умова спрощується до перевірки наявності активних зустрічних ордерів, оскільки ринковий ордер за визначенням погоджується на будь-яку ціну.

Обсяг угоди визначається як мінімум залишкових обсягів обох ордерів: $matchQty(o_1, o_2) = \min(remaining(o_1), remaining(o_2))$. Це правило гарантує, що жоден ордер не буде виконано на обсяг, більший за залишок. Якщо залишки не рівні, один з ордерів буде виконано повністю, а інший – частково.

Побудована формальна модель слугує специфікацією для обох реалізацій демонстраційної системи. Традиційна архітектура та архітектура на основі віртуальних акторів повинні реалізовувати однакову семантику операцій та підтримувати ідентичні інваріанти. Різниця між ними проявляється лише у способі забезпечення атомарності операцій: традиційний підхід використовує розподілені блокування, тоді як віртуальні актори покладаються на гарантію єдиної активації. Формальна модель дозволяє верифікувати коректність обох реалізацій шляхом перевірки інваріантів після кожної операції у тестових сценаріях.

2.2 Алгоритм співставлення ордерів

Алгоритм співставлення (matching engine) є ядром торгової системи та відповідає за пошук зустрічних ордерів, обчислення параметрів угоди та оновлення стану стакану. Коректність та ефективність цього алгоритму безпосередньо впливають на справедливість торгів та продуктивність системи. У цьому підрозділі формалізовано алгоритм, наведено оцінку його обчислювальної складності та доведено збереження інваріантів консистентності.

Співставлення ордерів відбувається за принципом price–time priority, описаним у попередньому підрозділі. Коли до системи надходить новий агресивний ордер (той, що ініціює угоду), алгоритм послідовно перевіряє найкращі зустрічні ордери на можливість співставлення. Для ордера на купівлю зустрічними є ордери на продаж з множини Asks, для ордера на продаж – ордери на купівлю з множини Bids. Співставлення продовжується доки виконуються дві умови: залишається невиконаний обсяг агресивного ордера та існують зустрічні ордери з прийнятною ціною.

Умова прийнятності ціни залежить від типу агресивного ордера. Для лімітного ордера на купівлю з ціною p_{buy} співставлення можливе з ордером на продаж за ціною p_{ask} , якщо $p_{buy} \geq p_{ask}$. Ця умова означає, що покупець готовий платити не менше, ніж просить продавець. Для лімітного ордера на продаж з ціною p_{sell} співставлення можливе з ордером на купівлю за ціною p_{bid} , якщо $p_{sell} \leq p_{bid}$. Ринкові ордери не мають цінового обмеження та співставляються з будь-яким доступним зустрічним ордером.

Ціна виконання угоди визначається ціною пасивного ордера, тобто того, що вже знаходився у стакані. Це правило називається maker–taker pricing і є стандартом для більшості бірж. Пасивний ордер (maker) надає ліквідність ринку, тому отримує гарантію виконання за своєю ціною. Агресивний ордер (taker) забирає ліквідність і погоджується на ціну maker'a. Таке правило стимулює розміщення лімітних ордерів, що збільшує глибину стакану та зменшує спред.

Обсяг кожної окремої угоди визначається як мінімум залишкових обсягів обох ордерів: $matchQty = \min(\text{remaining}(o_aggressive), \text{remaining}(o_passive))$. Після виконання угоди поле filled обох ордерів збільшується на matchQty. Якщо залишковий обсяг ордера стає нульовим, він вважається повністю виконаним і видаляється зі стакану.

Алгоритм обробки лімітного ордера на купівлю складається з трьох фаз: фази співставлення, фази розміщення та фази фіналізації. У фазі співставлення алгоритм ітеративно обробляє найкращі ордери на продаж, поки виконуються умови співставлення. На кожній ітерації створюється угода, оновлюються обсяги

обох ордерів, і якщо пасивний ордер повністю виконано, він видаляється з множини Asks. У фазі розміщення, якщо після співставлення залишився невиконаний обсяг агресивного ордера, він додається до множини Bids у відповідну позицію згідно з правилами впорядкування. У фазі фіналізації оновлюються статуси всіх задіяних ордерів та генеруються події для нотифікації клієнтів.

Обробка ринкового ордера відрізняється відсутністю цінового обмеження та відсутністю фази розміщення. Ринковий ордер або виконується повністю за рахунок доступної ліквідності, або виконується частково і скасовується на залишок, якщо ліквідності недостатньо. Ринковий ордер ніколи не розміщується у стакан, оскільки не має визначеної ціни.

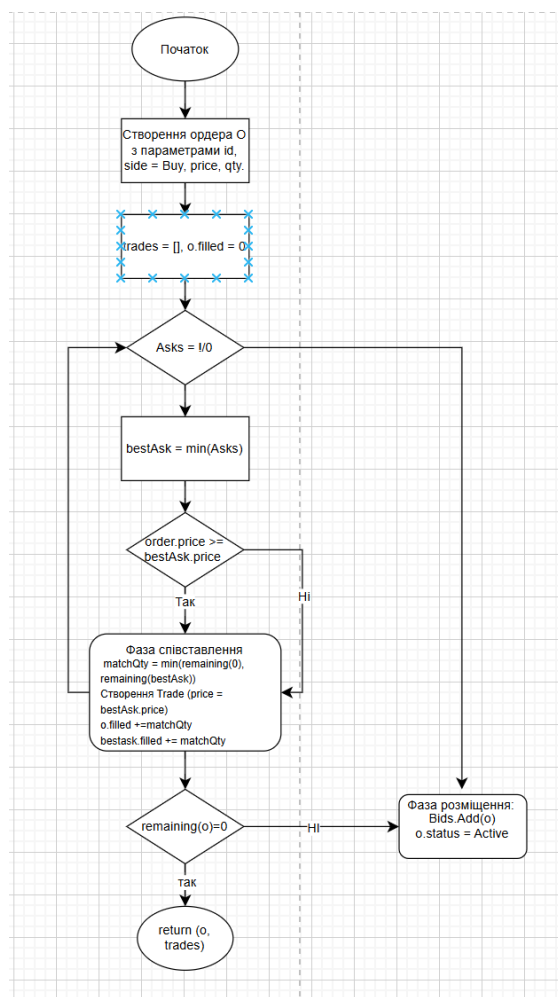


Рис. 2.3. Блок–схема алгоритму співставлення лімітного ордера на купівлю

Детальний псевдокод алгоритму наведено у додатку А. Тут розглянемо ключові аспекти реалізації та їх вплив на продуктивність.

Обчислювальна складність алгоритму залежить від структур даних, обраних для зберігання ордерів. У демонстраційній системі множини Bids та Asks реалізовані як збалансовані дерева пошуку (SortedList у .NET), що забезпечують логарифмічну складність операцій вставки та видалення. Пошук найкращого ордера виконується за константний час $O(1)$, оскільки він завжди знаходиться на початку впорядкованої колекції.

Складність обробки одного лімітного ордера оцінюється як $O(k \cdot \log n + \log n)$, де n є загальною кількістю ордерів у стакані, а k є кількістю угод, що виникли внаслідок співставлення. Перший доданок $k \cdot \log n$ відповідає фазі співставлення: кожна з k угод потребує видалення пасивного ордера зі складністю $O(\log n)$. Другий доданок $\log n$ відповідає фазі розміщення, якщо вона виконується. У типовому випадку k є невеликим числом (один–два ордери), тому практична складність близька до $O(\log n)$.

Для ринкових ордерів складність становить $O(k \cdot \log n)$, де k може бути значно більшим, оскільки великий ринковий ордер може "проїсти" багато рівнів стакану. У екстремальному випадку, коли ринковий ордер виконується проти всіх ордерів у стакані, складність зростає до $O(n \cdot \log n)$.

Коректність алгоритму визначається збереженням інваріантів, встановлених у підрозділі 2.1. Доведемо, що алгоритм підтримує інваріант I_2 (відсутність перетину цін).

Теорема 1 (збереження I_2). Нехай стакан ОВ знаходиться у стані, що задовольняє інваріант I_2 . Після виконання операції `AddLimitBuy(OB, o)` результуючий стакан ОВ' також задовольняє I_2 .

Доведення. Розглянемо два випадки. Випадок 1: ціна нового ордера `o.price` менша за `bestAsk`. Тоді умова співставлення не виконується, ордер додається до Bids без змін у Asks. Новий `bestBid'` може дорівнювати `o.price` або залишитись попереднім `bestBid`. В обох випадках $\text{bestBid}' \leq \text{o.price} < \text{bestAsk} = \text{bestAsk}'$, тому I_2 виконується. Випадок 2: ціна `o.price` не менша за `bestAsk`. Тоді виконується

співставлення, яке видаляє ордери з Asks доки $o.price \geq bestAsk$ або доки не вичерпано обсяг o . Якщо після співставлення Asks стає порожньою, I_2 тривіально виконується. Якщо Asks не порожня, то новий $bestAsk' > o.price \geq$ старий $bestAsk$. Якщо o повністю виконано, $bestBid' =$ попередній $bestBid <$ попередній $bestAsk \leq bestAsk'$, тому I_2 виконується. Якщо o частково виконано і розміщено у Bids, то $bestBid' \leq o.price < bestAsk'$ за умовою виходу з циклу співставлення. Таким чином, у всіх випадках I_2 зберігається.

Аналогічні доведення можна провести для інших операцій та інваріантів. Формальна верифікація гарантує, що обидві реалізації – традиційна та на основі Orleans – при коректному виконанні алгоритму підтримуватимуть консистентний стан стакану.

Атомарність операцій є критичною для коректності алгоритму у конкурентному середовищі. Весь цикл обробки ордера – від перевірки умови співставлення до оновлення стану – повинен виконуватись як неподільна операція. У традиційній архітектурі атомарність забезпечується утриманням розподіленого блокування протягом всієї операції. В архітектурі віртуальних акторів атомарність гарантується моделлю послідовного виконання: грейн обробляє один запит за раз, і наступний запит починає виконуватись лише після завершення попереднього. Ця відмінність є ключовою для розуміння різниці у продуктивності, яку демонструють експерименти.

2.3 Архітектура демонстраційної системи

Для проведення експериментального дослідження спроектовано та реалізовано демонстраційну систему, яка дозволяє порівняти традиційну архітектуру з розподіленими блокуваннями та архітектуру на основі віртуальних акторів в ідентичних умовах. Ключовим проектним рішенням є реалізація обох підходів з однаковою бізнес-логікою та єдиним зовнішнім інтерфейсом, що забезпечує коректність порівняння та дозволяє використовувати спільний код для тестування.

Система побудована за принципом модульності та складається з шести .NET проектів і клієнтського додатку на Angular. Така структура забезпечує чітке

розділення відповідальностей: спільні моделі даних винесено в окремий проект, кожна архітектурна реалізація ізольована у власному проекті, модуль тестування не залежить від деталей реалізації серверів. Взаємозв'язки між компонентами системи представлено на рисунку 2.4.

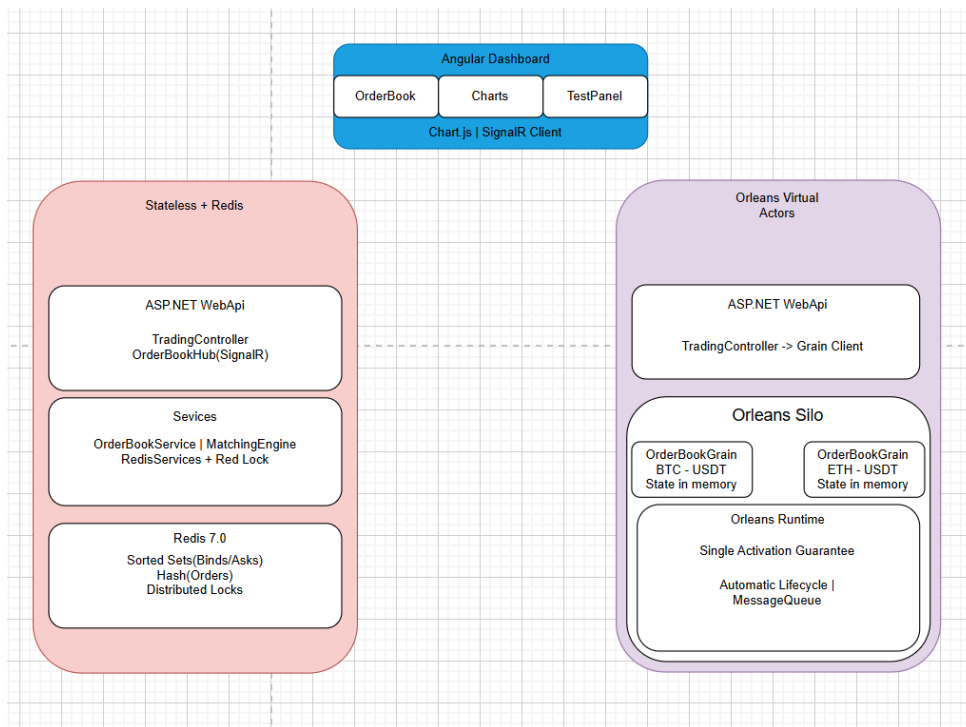


Рис. 2.4. Архітектура демонстраційної системи та взаємодія компонентів

Проект **TradingDemo.Shared** містить спільні моделі даних та інтерфейси, які використовуються обома серверними реалізаціями та клієнтським кодом. До цього проекту належать класи **Order**, **Trade**, **OrderBookSnapshot** та **PerformanceMetrics**, описані у формальній моделі. Винесення моделей у спільний проект гарантує консистентність структур даних та дозволяє серіалізувати об'єкти на одній стороні і десеріалізувати на іншій без конфліктів. Усі моделі позначені атрибутами серіалізації Orleans, що забезпечує їх коректну передачу між грейнами та клієнтами.

Проект **TradingDemo.Stateless** реалізує традиційну архітектуру без збереження стану на сервері додатку. Стан торгового стакана зберігається у зовнішньому сховищі **Redis**, а консистентність при конкурентному доступі забезпечується механізмом розподілених блокувань. Проект містить контролер **TradingController** для обробки HTTP-запитів, сервіс **OrderBookService** з бізнес-

логікою торгового стакану, сервіс `MatchingEngine` з алгоритмом співставлення та сервіс `RedisService` для взаємодії зі сховищем. Додатково реалізовано `OrderBookHub` для передачі оновлень стакану клієнтам через `SignalR` у реальному часі.

Проект `TradingDemo.Orleans` реалізує архітектуру на основі віртуальних акторів та має внутрішню структуру з трьох підпроектів. Підпроект `Grains` містить інтерфейс `IOrderBookGrain` та його реалізацію `OrderBookGrain`, які інкапсулюють стан стакану та логіку обробки ордерів. Підпроект `Silo` відповідає за конфігурацію та запуск серверного процесу `Orleans`, включаючи налаштування кластеризації та провайдерів збереження стану. Підпроект `WebApi` надає `REST API`, ідентичний до `Stateless` версії, та виступає точкою входу для клієнтських запитів, які транслуються у виклики методів грейнів.

Проект `TradingDemo.LoadTesting` містить сценарії навантажувального тестування, реалізовані з використанням бібліотеки `NBomber`. Кожен сценарій оформлено як окремий клас: `ConcurrentOrdersTest` для тестування максимальної конкуренції, `MarketCrashTest` для симуляції обвалу ринку, `HighFrequencyTest` для тривалого високочастотного навантаження. Модуль тестування взаємодіє з обома серверами через `HTTP API`, що дозволяє виконувати ідентичні тести на обох реалізаціях та порівнювати результати.

Клієнтський додаток реалізовано як односторінковий веб-застосунок на `Angular 20` з використанням `TypeScript`. Компонент `order-book-panel` відображає візуалізацію торгового стакану з рівнями цін на купівлю та продаж, оновлюючись у реальному часі через `SignalR`. Компонент `performance-chart` візуалізує метрики продуктивності за допомогою бібліотеки `Chart.js`, включаючи графіки латентності, пропускної здатності та розподілу помилок. Компонент `test-control-panel` надає інтерфейс для вибору та запуску тестових сценаріїв. Компонент `dashboard` об'єднує всі елементи в єдиний інтерфейс моніторингу, що відображає два стакани поруч для наочного порівняння.

Технологічний стек обрано з урахуванням сучасних практик розробки високопродуктивних систем та вимог до відтворюваності експериментів.

Серверна частина базується на ASP.NET Core 8.0 – довгостроково підтримуваний версії [7] з найкращою продуктивністю серед .NET веб-фреймворків. Для реалізації віртуальних акторів використовується Microsoft Orleans 8.0, повністю сумісний з .NET 8 та оптимізований для високонавантажених сценаріїв.

Традиційна реалізація використовує Redis 7.0 як сховище стану. Redis обрано через виняткову швидкодію операцій в оперативній пам'яті, вбудовану підтримку Sorted Sets для зберігання впорядкованих колекцій ордерів та наявність примітивів для реалізації розподілених блокувань. Взаємодія з Redis здійснюється через бібліотеку StackExchange.Redis, а розподілені блокування реалізовано за допомогою RedLock.net, що імплементує алгоритм Redlock для забезпечення коректності блокувань [33] у розподіленому середовищі.

Orleans реалізація використовує вбудоване збереження стану в пам'яті грейна, що є достатнім для демонстраційних цілей та дозволяє ізолювати вплив дискових операцій на результати вимірювань. При потребі персистентності Orleans підтримує різні провайдери збереження, включаючи Azure Storage, AWS DynamoDB та реляційні бази даних через ADO.NET.

Бібліотека NBomber 5.0 використовується для генерації навантаження [35] та збору метрик. NBomber надає декларативний API для опису сценаріїв тестування, автоматичний розрахунок перцентилів латентності та можливість генерації детальних HTML-звітів. Бібліотека підтримує моделювання різних патернів навантаження: константного, зростаючого та імпульсного.

Комунікація між клієнтом та серверами здійснюється через два канали. HTTP використовується для разових запитів, таких як розміщення ордерів та отримання знімків стакану. SignalR забезпечує двосторонній зв'язок [36] у реальному часі для передачі оновлень стакану та метрик продуктивності без необхідності періодичного опитування.

Усі компоненти системи упаковано в Docker-контейнери, що спрощує розгортання та забезпечує відтворюваність результатів експериментів на різних середовищах. Docker Compose використовується для оркестрації контейнерів та налаштування мережевої взаємодії між ними. Конфігурація включає контейнери

для Redis, Orleans Silo, Stateless API, клієнтського додатку та опціонально для інструментів моніторингу.

Обидві реалізації надають ідентичний REST API з шести ендпоінтів, що забезпечує справедливе порівняння та дозволяє використовувати єдиний клієнтський код. Ендпоінт POST /api/orders/limit приймає параметри symbol, side, price та quantity і повертає створений ордер разом зі списком угод, якщо відбулось співставлення. Ендпоінт POST /api/orders/market обробляє ринкові ордери без вказання ціни. Ендпоінт GET /api/orderbook/{symbol} повертає знімок стакану з агрегованими рівнями цін. Ендпоінт DELETE /api/orders/{orderId} скасовує активний ордер. Ендпоінт GET /api/trades/{symbol}/recent повертає історію останніх угод. Ендпоінт GET /api/metrics повертає поточні показники продуктивності, включаючи специфічні для кожної реалізації метрики блокувань або грейнів.

Повний код реалізації наведено у додатку Б. Структура каталогів проекту відповідає стандартам організації .NET рішень та забезпечує зручну навігацію між компонентами.

2.4 Порівняльна реалізація архітектурних підходів

Для наочної демонстрації відмінностей між традиційною архітектурою та архітектурою віртуальних акторів обидві реалізації побудовано з ідентичною бізнес-логікою, але принципово різними механізмами забезпечення консистентності. У цьому підрозділі проаналізовано ключові відмінності на рівні коду та оцінено їх вплив на складність розробки та супроводу.

Традиційна реалізація базується на патерні stateless-сервісів, де сервер додатку не зберігає стан між запитами. Весь стан торгового стакану винесено у зовнішнє сховище Redis, що забезпечує можливість горизонтального масштабування серверного рівня та відмовостійкість при перезапуску окремих екземплярів. Проте ця архітектура створює додаткові виклики: кожна операція потребує звернення до мережевого сховища, а конкурентний доступ до спільного ресурсу вимагає механізмів синхронізації.

Для зберігання ордерів у Redis використовуються структури Sorted Set, які забезпечують автоматичне впорядкування елементів за числовим score та ефективні операції вставки та видалення зі складністю $O(\log n)$. Ордери на купівлю зберігаються у ключі `book:{symbol}:bids` зі score, що дорівнює від'ємній ціні, що забезпечує сортування від найвищої ціни до найнижчої. Ордери на продаж зберігаються у ключі `book:{symbol}:asks` з додатнім score для сортування від найнижчої ціни до найвищої. Деталі кожного ордера серіалізуються у JSON та зберігаються в окремих ключах `order:{symbol}:{orderId}`.

Критичним елементом традиційної реалізації є механізм розподілених блокувань на основі алгоритму Redlock. Перед модифікацією стакана сервіс намагається отримати ексклюзивне блокування на ключ `lock:orderbook:{symbol}`. Алгоритм використовує команду SET з параметрами NX (встановити лише якщо ключ не існує) та PX (час життя у мілісекундах). Якщо блокування зайняте іншим процесом, поточний процес очікує у циклі з періодичними повторними спробами до досягнення таймауту.

Звільнення блокування реалізовано через атомарний Lua-скрипт, який перевіряє, що поточний процес є власником блокування перед його видаленням. Без цієї перевірки можлива ситуація, коли процес А, чий час блокування вичерпався, випадково видаляє блокування процесу Б. Lua-скрипт виконується атомарно на стороні Redis, що гарантує коректність операції.

Обробка лімітного ордера у традиційній реалізації включає наступну послідовність дій. Спочатку сервіс намагається отримати блокування з таймаутом п'ять секунд. Якщо блокування не вдалося отримати, ордер відхиляється з помилкою таймауту, а лічильник втрачених ордерів інкрементується. Після успішного отримання блокування сервіс зчитує поточний стан стакана з Redis, виконує алгоритм співставлення локально, записує оновлений стан назад у Redis та звільняє блокування у блоці finally для гарантії звільнення навіть при виникненні виключення.

Реалізація на основі Orleans радикально спрощує код завдяки гарантіям, що надаються фреймворком. Грейн `OrderBookGrain` інкапсулює стан стакана у

приватних полях класу: SortedList для Bids з компаратором спадання ціни, SortedList для Asks зі стандартним компаратором, List для збереження угод та Dictionary для швидкого доступу до активних ордерів за ідентифікатором. Стан зберігається безпосередньо в оперативній пам'яті сервера, на якому активовано грейн, що усуває мережеву затримку при кожній операції.

Гарантія єдиної активації [11] Orleans забезпечує, що у будь-який момент часу існує щонайбільше один екземпляр грейна з заданим ідентифікатором у всьому кластері. Це означає, що всі запити до конкретного торгового стакану (наприклад, "BTC-USD") автоматично направляються до одного грейна. Середовище виконання Orleans організує чергу запитів та виконує їх послідовно, один за одним. Наступний запит починає виконуватись лише після повного завершення попереднього, включаючи всі асинхронні операції.

Така модель виконання повністю усуває потребу у явних механізмах синхронізації. Код методу PlaceLimitOrderAsync у Orleans версії не містить жодних блокувань, повторних спроб чи обробки таймаутів. Метод просто створює ордер, викликає алгоритм співставлення, оновлює внутрішній стан та повертає результат. Коректність гарантується архітектурою фреймворку, а не дисципліною розробника.

Ініціалізація грейна виконується автоматично при першому зверненні. Метод OnActivateAsync викликається фреймворком і дозволяє виконати початкове налаштування, зокрема ініціалізацію колекцій з відповідними компараторами. Деактивація грейна при тривалій неактивності також відбувається автоматично, звільняючи ресурси для інших грейнів. Розробнику не потрібно явно керувати життєвим циклом.

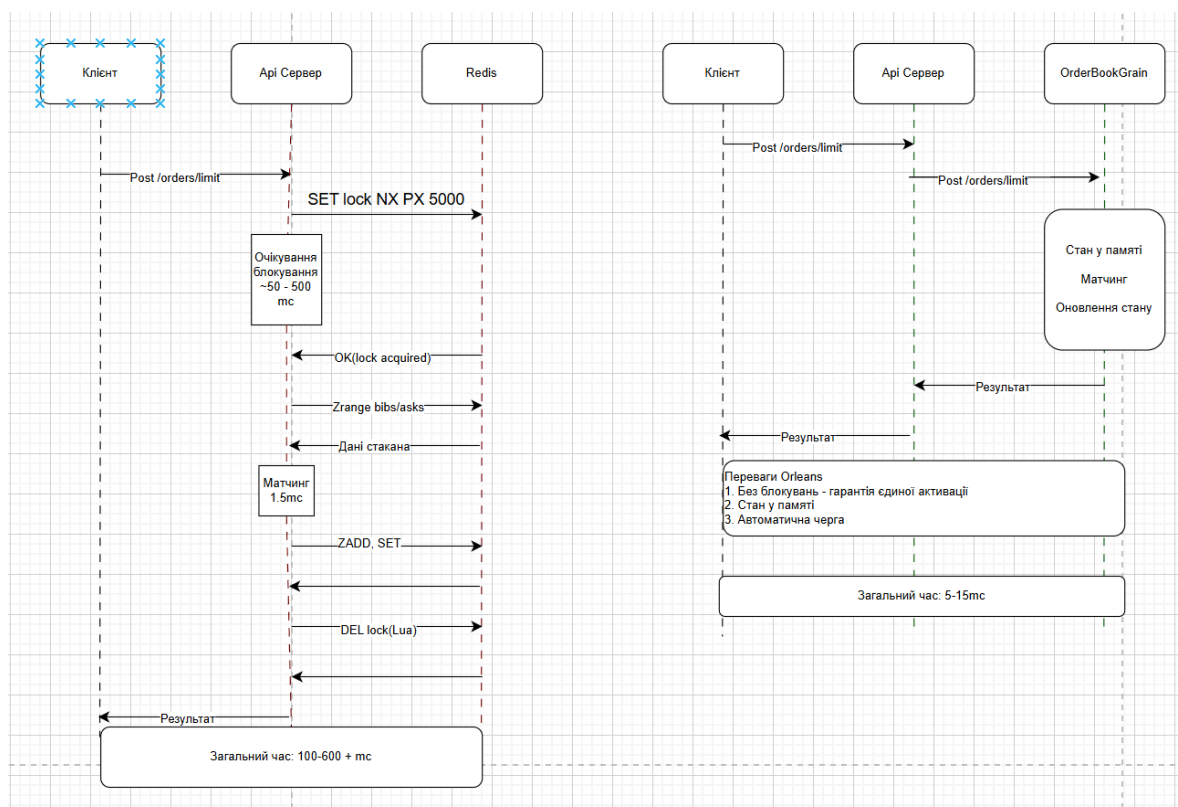


Рис. 2.5. Порівняння потоків виконання: традиційна архітектура та віртуальні актори

Порівняльний аналіз обсягу коду демонструє суттєве спрощення при використанні Orleans. Метод `PlaceLimitOrderAsync` у традиційній реалізації містить приблизно шістдесят рядків коду, з яких близько двадцяти п'яти відповідають за логіку блокувань, повторних спроб та обробку таймаутів. Ще близько п'ятнадцяти рядків реалізують серіалізацію та взаємодію з Redis. Власне бізнес-логіка співставлення займає лише двадцять рядків. Аналогічний метод у Orleans версії містить приблизно тридцять п'ять рядків, майже повністю присвячених бізнес-логіці.

Відсутність інфраструктурного коду у Orleans версії має глибші наслідки, ніж просто менша кількість рядків. Код традиційної реалізації містить множину потенційних точок відмови: невдале отримання блокування, таймаут з'єднання з Redis, помилка серіалізації, втрата блокування під час операції. Кожна з цих ситуацій потребує окремої обробки та тестування. Orleans версія архітектурно виключає більшість цих сценаріїв.

Для кількісної оцінки складності реалізацій застосовано метрики, наведені у таблиці 2.2. Рядки коду (LOC) підраховано для файлів, що безпосередньо реалізують логіку торгового стакана, без урахування конфігурації та допоміжного коду. Цикломатична складність оцінює кількість незалежних шляхів виконання у коді та відображає когнітивне навантаження при читанні та супроводі. Кількість залежностей враховує зовнішні бібліотеки, необхідні для функціонування кожної реалізації.

Таблиця 2.2

Метрики складності реалізацій

Метрика	Stateless + Redis	Orleans	Різниця
Рядки коду (LOC)	580	340	–41%
Цикломатична складність	45	18	–60%
Кількість класів	8	4	–50%
Зовнішні залежності	4	2	–50%
Код синхронізації (LOC)	120	0	–100%

Результати аналізу підтверджують теоретичні очікування: модель віртуальних акторів дозволяє зосередитись на бізнес-логіці, делегуючи інфраструктурні аспекти фреймворку. Зменшення цикломатичної складності на 60% означає суттєве спрощення тестування та зниження ймовірності помилок. Повна відсутність коду синхронізації усуває цілий клас потенційних дефектів, пов'язаних з deadlocks, race conditions та некоректним звільненням блокувань.

Важливо відзначити, що спрощення коду не означає втрати функціональності чи контролю. Orleans надає механізми для реалізації складних сценаріїв: реєстрабельні грейни для вкладених викликів, таймери та нагадування для відкладених операцій, стріми для реактивної обробки подій. Проте для типового випадку обробки торгових ордерів базова модель послідовного виконання є достатньою та оптимальною.

Детальний код обох реалізацій наведено у додатку Б. Порівняння конкретних фрагментів дозволяє наочно побачити різницю у підходах та оцінити переваги кожного з них для різних сценаріїв застосування.

2.5 Методика експериментального дослідження

Для отримання достовірних результатів порівняння архітектурних підходів розроблено методику експериментального дослідження, яка визначає умови проведення експериментів, тестові сценарії, метрики оцінки та критерії прийняття гіпотез. Методика побудована таким чином, щоб мінімізувати вплив зовнішніх факторів та забезпечити відтворюваність результатів.

Експерименти проводяться на одному фізичному сервері з конфігурацією, достатньою для виявлення різниці між підходами без штучних обмежень ресурсів. Використання одного сервера замість розподіленого кластера є свідомим рішенням, що дозволяє ізолювати архітектурні відмінності від впливу мережевої затримки між вузлами. У реальних системах мережева затримка додає константну складову до часу обробки обох підходів, але не змінює співвідношення між ними при високій конкуренції.

Конфігурація тестового середовища включає процесор з вісьмома ядрами для забезпечення паралелізму при генерації навантаження, шістнадцять гігабайт оперативної пам'яті для розміщення стану грейнів та кешу Redis, та SSD-накопичувач для мінімізації впливу дискових операцій на результати. Операційна система Ubuntu 22.04 LTS обрана як стабільна платформа з передбачуваною поведінкою. Усі компоненти системи запускаються в Docker-контейнерах з обмеженням ресурсів для забезпечення ідентичних умов між запусками.

Перед кожним тестом виконується процедура підготовки, яка включає перезапуск усіх контейнерів для очищення стану, прогрівання системи серією з п'ятисот ордерів для активації грейнів та заповнення кешів, та паузу у десять секунд для стабілізації. Результати прогрівання не враховуються у фінальних метриках. Кожен тестовий сценарій виконується п'ять разів, після чого обчислюються середні значення та стандартне відхилення для оцінки стабільності результатів.

Для демонстрації різниці між архітектурами розроблено три сценарії навантаження, кожен з яких акцентує певний аспект поведінки систем під навантаженням.

Сценарій "Concurrent Orders" моделює ситуацію екстремальної конкуренції, коли велика кількість ордерів одночасно намагається модифікувати один ресурс. Тисяча лімітних ордерів на купівлю за однаковою ціною надсилаються паралельно з мінімальною затримкою між запитами. Усі ордери конкурують за один рівень ціни у стакані, що створює максимальний тиск на механізми синхронізації. Для традиційної архітектури цей сценарій є найскладнішим: лише один процес може утримувати блокування у кожен момент часу, решта очікує у черзі. Для Orleans усі запити направляються до одного грейна, який обробляє їх послідовно без накладних витрат на блокування. Ключовою метрикою є надійність обробки – відсоток успішно оброблених ордерів без втрат через таймаути.

Сценарій "Market Crash" моделює різкий рух ринку, коли великий ринковий ордер "проїдає" значну частину стакана. Спочатку стакан наповнюється п'ятьмастами лімітними ордерами на продаж, розподіленими по двадцяти рівнях ціни. Потім надсилається ринковий ордер на купівлю з обсягом, що перевищує сумарний обсяг верхніх десяти рівнів. Ордер повинен послідовно виконатись проти множини рівнів за одну атомарну операцію. Сценарій тестує коректність алгоритму часткового виконання та атомарність операцій. Ключовою метрикою є консистентність – відповідність фінального стану стакана очікуваному після виконання всіх угод.

Сценарій "High Frequency" моделює тривале високочастотне навантаження, типове для реальних торгових систем. Протягом шістдесяти секунд генерується постійний потік ордерів з інтенсивністю тисяча ордерів на секунду. Розподіл операцій відповідає реалістичному профілю: сорок відсотків лімітних ордерів на купівлю, сорок відсотків лімітних на продаж, десять відсотків ринкових та десять відсотків скасування. Ціни та обсяги генеруються випадково у визначених діапазонах. Сценарій дозволяє оцінити стабільність

системи під тривалим навантаженням та виявити деградацію продуктивності з часом. Ключовими метриками є пропускна здатність та латентність на різних перцентилях.

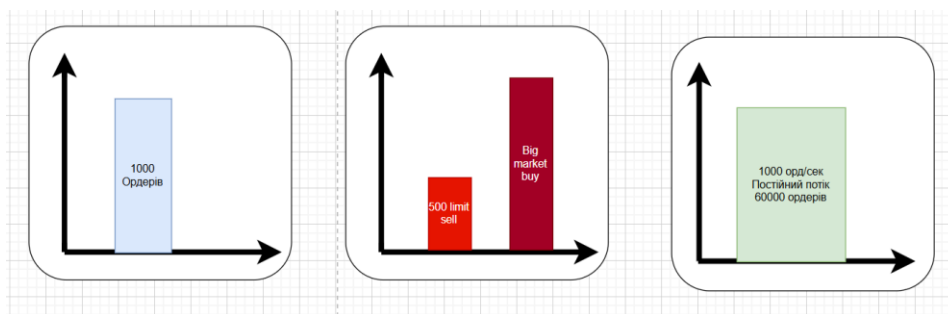


Рис. 2.6. Профілі навантаження тестових сценаріїв

Метрики продуктивності збираються на двох рівнях: клієнтському та серверному. Клієнтські метрики вимірюють час від відправлення HTTP-запиту до отримання відповіді, включаючи мережеву затримку та час обробки на сервері. Серверні метрики вимірюються всередині коду обробки через таймери Stopwatch та відображають чистий час виконання бізнес-логіки. Для аналізу використовуються переважно серверні метрики, оскільки вони точніше відображають архітектурні відмінності.

Пропускна здатність (throughput) визначається як кількість успішно оброблених ордерів за секунду. Метрика обчислюється як відношення загальної кількості успішних операцій до тривалості тесту. Для сценарію "Concurrent Orders" обчислюється миттєва пропускна здатність як кількість ордерів, поділена на час обробки всієї партії.

Латентність вимірюється для кожної операції окремо та агрегується у перцентилі. Медіана (p50) характеризує типовий час обробки. Дев'яносто п'ятий перцентиль (p95) показує час, за який виконується дев'яносто п'ять відсотків операцій. Дев'яносто дев'ятий перцентиль (p99) є критичним для торгових систем, оскільки характеризує найгірші випадки, що безпосередньо впливають на досвід користувачів. Максимальне значення фіксує екстремальні викиди.

Надійність обробки визначається як відношення кількості успішних операцій до загальної кількості спроб, виражене у відсотках. Для традиційної архітектури окремо підраховуються втрати через таймаути блокування та

помилки з'єднання з Redis. Для Orleans втрати можливі лише через виключення у бізнес-логіці, що при коректній реалізації не повинно відбуватись.

Для традиційної реалізації додатково збираються метрики блокувань: середній час очікування блокування, максимальний час очікування, кількість випадків конкуренції (коли блокування було зайняте при першій спробі), кількість таймаутів. Ці метрики дозволяють кількісно оцінити накладні витрати механізму синхронізації.

На основі теоретичного аналізу та попередніх досліджень сформульовано чотири гіпотези, які перевіряються експериментально.

Гіпотеза H1 стверджує, що реалізація на основі віртуальних акторів забезпечує вищу пропускну здатність при високій конкуренції за спільний ресурс. Кількісний критерій: $\Theta_Orleans > 2 \times \Theta_Stateless$ при одночасній обробці понад п'ятисот ордерів на один стакан. Обґрунтування: Orleans усуває накладні витрати на мережеві запити до Redis та очікування блокувань.

Гіпотеза H2 стверджує, що латентність на перцентилі p99 для віртуальних акторів є суттєво нижчою та стабільнішою. Кількісний критерій: $\lambda_p99_Orleans < 0.1 \times \lambda_p99_Stateless$ при навантаженні понад тисячу ордерів на секунду. Обґрунтування: відсутність черги очікування блокування усуває довгий "хвіст" розподілу латентності.

Гіпотеза H3 стверджує, що надійність обробки для віртуальних акторів становить сто відсотків незалежно від інтенсивності навантаження. Кількісний критерій: $\rho_Orleans = 100\%$ для всіх сценаріїв. Обґрунтування: архітектура Orleans не передбачає таймаутів на рівні моделі виконання; грейн обробляє всі запити з черги послідовно.

Гіпотеза H4 стверджує, що складність реалізації бізнес-логіки для віртуальних акторів є нижчою. Кількісний критерій: $LOC_Orleans < 0.7 \times LOC_Stateless$ та повна відсутність коду синхронізації. Обґрунтування: модель віртуальних акторів делегує інфраструктурні аспекти фреймворку.

Для кожної гіпотези визначено критерій прийняття: гіпотеза вважається підтвердженою, якщо результати всіх п'яти повторень експерименту

задовольняють кількісний критерій. Якщо хоча б один результат не відповідає критерію, гіпотеза вважається частково підтвердженою або спростованою залежно від величини відхилення.

Результати експериментів фіксуються у структурованому форматі JSON для подальшого аналізу та візуалізації. Автоматично генеруються HTML-звіти з графіками розподілу латентності, динаміки пропускну здатності та порівняльними таблицями. Усі вихідні дані зберігаються для забезпечення можливості незалежної верифікації результатів.

Висновки до другого розділу

У розділі спроектовано та реалізовано демонстраційну систему для порівняння архітектур.

Побудовано формальну модель торгового стакану. Структура включає списки Bids та Asks, впорядковані за ціною та часом. Ключовий інваріант – $\text{bestBid} < \text{bestAsk}$. Він гарантує неможливість арбітражу.

Розроблено алгоритм матчингу ордерів за принципом price-time priority. Алгоритм підтримує часткове виконання. Обчислювальна складність – $O(k \cdot \log n)$.

Архітектура системи включає 6 .NET-проектів та Angular-клієнт. Реалізовано дві версії з ідентичним REST API. Stateless версія використовує Redis та RedLock. Orleans версія використовує грейни та силос. Ідентичний API забезпечує коректність порівняння.

Порівняння коду показало суттєву різницю. Orleans версія містить на 41% менше рядків. Цикломатична складність нижча на 60%. Код синхронізації відсутній повністю.

Розроблено методику тестування з трьома сценаріями. Concurrent Orders – 1000 одночасних ордерів на одну ціну. Market Crash – великий ринковий ордер проти множини менших. High Frequency – 60 000 ордерів за хвилину.

РОЗДІЛ 3. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

3.1 Вимоги до системи та середовище виконання

Для коректної роботи демонстраційної системи та відтворюваності результатів експериментів визначено мінімальні та рекомендовані вимоги до апаратного та програмного забезпечення. Вибір конфігурації обумовлений необхідністю забезпечити достатні ресурси для одночасної роботи обох архітектурних реалізацій без штучних обмежень, які могли б спотворити результати порівняння.

Апаратні вимоги до системи визначаються характером навантажувального тестування, яке передбачає генерацію тисяч конкурентних запитів та їх паралельну обробку. Мінімальна конфігурація включає чотириядерний процесор з тактовою частотою не менше 2.5 ГГц, вісім гігабайтів оперативної пам'яті та твердотільний накопичувач для забезпечення швидкого доступу до даних. Рекомендована конфігурація передбачає восьмиядерний процесор для ефективної паралельної генерації навантаження, шістнадцять гігабайтів оперативної пам'яті для комфортної роботи всіх компонентів системи та NVMe накопичувач для мінімізації затримок вводу–виводу.

Експерименти, результати яких наведено у даній роботі, проводились на робочій станції з процесором AMD Ryzen 7 (вісім ядер), тридцятьма двома гігабайтами оперативної пам'яті DDR5 та SSD накопичувачем об'ємом п'ятсот дванадцять гігабайтів. Використання локального середовища замість хмарної інфраструктури забезпечило повний контроль над умовами експерименту та виключило вплив мережеских затримок та варіативності продуктивності віртуальних машин.

Програмне середовище базується на платформі .NET 8.0, яка є довгостроково підтримуваною версією (LTS) з оптимізаціями продуктивності та покращеною підтримкою асинхронних операцій. Операційна система Windows 11 Pro обрана як основна платформа розробки, проте система повністю сумісна

з Linux завдяки кросплатформності .NET. Контейнеризація забезпечується Docker Desktop версії 4.25 з підтримкою WSL2 для ефективної роботи Linux–контейнерів у Windows–середовищі.

Таблиця 3.1

Конфігурація програмного середовища

Компонент	Версія	Призначення
.NET Runtime	8.0 LTS	Середовище виконання серверів
ASP.NET Core	8.0	Веб–фреймворк для REST API
Microsoft Orleans	8.0	Фреймворк віртуальних акторів
Redis	7.0	Сховище стану для Stateless версії
Node.js	20 LTS	Середовище для Angular CLI
Angular	20	Фреймворк клієнтського додатку
Docker Desktop	4.25	Контейнеризація компонентів

Архітектура розгортання передбачає ізоляцію компонентів у окремих Docker–контейнерах для забезпечення відтворюваності та незалежності від локального середовища. Redis для традиційної архітектури запускається в окремому контейнері з виділенням двох гігабайтів оперативної пам'яті та налаштуванням persistence у режимі RDB для періодичного збереження стану. Контейнер Stateless API базується на образі `mcr.microsoft.com/dotnet/aspnet:8.0` з відкритим портом 5001 для HTTP–запитів та портом для SignalR–з'єднань. Контейнер Orleans Silo використовує той самий базовий образ з додатковими портами для внутрішньокластерної комунікації та gateway для клієнтських підключень. Веб–API Orleans працює на порту 5002.

Для забезпечення справедливого порівняння обидва серверні контейнери отримують ідентичні ресурсні обмеження: два ядра процесора та чотири гігабайти оперативної пам'яті. Такі обмеження запобігають ситуації, коли один з серверів отримує перевагу за рахунок захоплення більшої частки системних ресурсів. Клієнтський Angular–додаток виконується у браузері Chrome з вимкненими розширеннями для мінімізації стороннього впливу на результати вимірювань.

Мережева конфігурація Docker використовує bridge–мережу для з'єднання контейнерів між собою та з хост–системою. Усі компоненти знаходяться в одній мережі, що мінімізує мережеву латентність та наближує умови до розгортання на одному сервері. Для production–середовища рекомендується використання overlay–мережі при розгортанні на кластері Docker Swarm або Kubernetes.

Процедура підготовки середовища до експерименту включає кілька обов'язкових кроків. Перезапуск усіх контейнерів виконується командою `docker-compose down` з подальшим `docker-compose up` для гарантованого очищення стану. Прогрівальний період тривалістю тридцять секунд дозволяє JIT–компілятору .NET оптимізувати критичні шляхи виконання, а garbage collector – стабілізувати роботу. Під час прогріву генерується п'ятсот тестових ордерів, результати яких не враховуються у фінальних метриках. Пауза у десять секунд після прогріву забезпечує завершення фонових процесів та стабілізацію системи перед початком вимірювань.

Збір метрик здійснюється на трьох рівнях. Клієнтські метрики вимірюють повний час відгуку від надсилання HTTP–запиту до отримання відповіді, що відображає реальний досвід користувача. Серверні метрики деталізують внутрішню обробку: час очікування блокування для традиційної версії, час виконання алгоритму співставлення, час серіалізації та десеріалізації даних. Системні метрики охоплюють використання процесора та оперативної пам'яті кожним контейнером, які збираються через Docker Stats API.

Для статистичної достовірності кожен експеримент виконується п'ять разів з обчисленням середніх значень та стандартного відхилення. Використання перцентилів (p50, p95, p99) замість середнього арифметичного дозволяє краще оцінити розподіл часу відгуку та виявити аномальні затримки, які можуть бути приховані при усередненні. Дев'яносто дев'ятий перцентиль є особливо важливим для торгових систем, оскільки характеризує найгірші випадки, що безпосередньо впливають на досвід користувачів у моменти пікового навантаження.

3.2 Інтерфейс користувача та засоби візуалізації

Для проведення експериментів та наочної демонстрації різниці між архітектурними підходами розроблено веб-додаток з інтерактивним інтерфейсом. Клієнтська частина реалізована на Angular 20з використанням TypeScript та бібліотеки Chart.js для побудови графіків. Інтерфейс забезпечує одночасне відображення стану обох торгових систем, панель керування тестовими сценаріями та візуалізацію метрик продуктивності у реальному часі.

Головний екран додатку (рис. 3.1) складається з трьох функціональних зон: панелі тестування у верхній частині, двох торгових стаканів у центральній частині та області метрик у нижній частині. Така компоновка дозволяє одночасно спостерігати за поведінкою обох систем під час тестування та оперативно порівнювати результати.

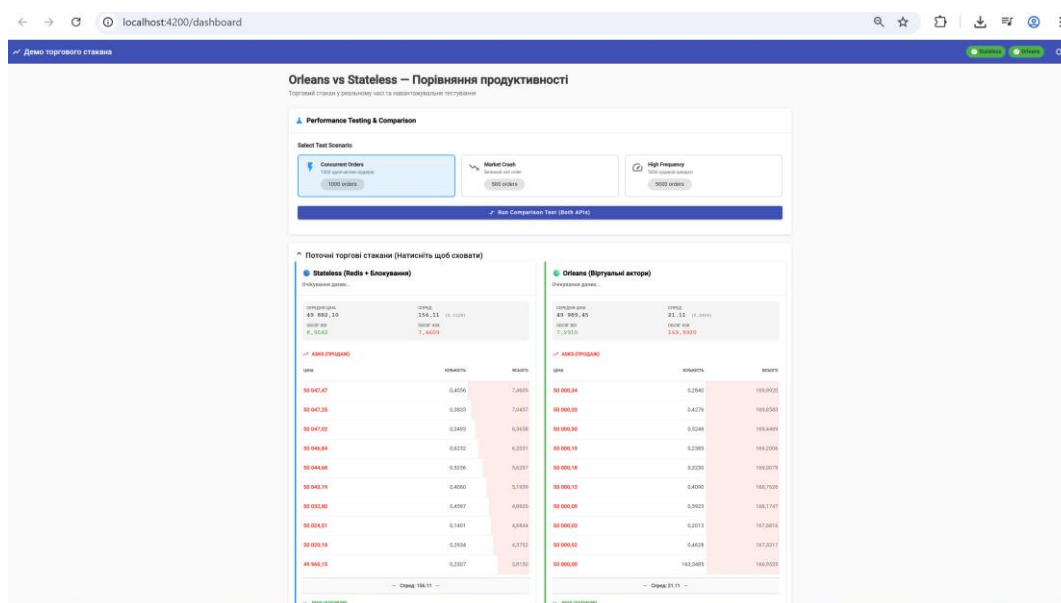


Рис. 3.1. Головний екран демонстраційної системи

Панель тестування розташована у верхній частині екрану та містить три кнопки вибору сценарію: Concurrent Orders для тестування максимальної конкуренції тисячею одночасних ордерів, Market Crash для симуляції обвалу ринку з великим ринковим ордером, та High Frequency для тривалого високочастотного навантаження п'ятьма тисячами ордерів. Кожна кнопка відображає короткий опис сценарію та кількість ордерів. Активний сценарій виділяється кольором для наочності. Кнопка "Run Comparison Test (Both APIs)"

запускає обраний сценарій одночасно на обох серверах, що забезпечує ідентичні умови тестування.

У правому верхньому куті розташовані індикатори стану з'єднання з серверами. Зелений індикатор Stateless свідчить про активне з'єднання з традиційним сервером на порту 5001, зелений індикатор Orleans – про з'єднання з сервером віртуальних акторів на порту 5002. При втраті з'єднання індикатор змінює колір на червоний, що дозволяє оперативно діагностувати проблеми.

Центральна частина екрану містить два торгових стакани, розташованих поруч для зручного візуального порівняння. Лівий стакан відображає стан системи Stateless (Redis + Locks), правий – Orleans (Virtual Actors). Кожен стакан містить заголовок з назвою архітектури та ключові метрики: Mid Price (середня ціна між найкращим bid та ask), Spread (різниця між найкращими цінами купівлі та продажу у абсолютному значенні та відсотках), Bid Volume (сумарний обсяг ордерів на купівлю) та Ask Volume (сумарний обсяг ордерів на продаж).

Таблиця ордерів на продаж (Asks) відображається червоним кольором у верхній частині кожного стакана. Ордери відсортовані за ціною від найнижчої (найкращої для покупця) до найвищої. Для кожного цінового рівня відображається ціна, обсяг та кумулятивний обсяг (Total), який показує сумарну кількість активу, доступну для купівлі до цього рівня включно. Горизонтальні смуги візуалізують відносний обсяг на кожному рівні, що дозволяє швидко оцінити розподіл ліквідності.

Таблиця ордерів на купівлю (Bids) відображається зеленим кольором у нижній частині стакана. Ордери відсортовані за ціною від найвищої (найкращої для продавця) до найнижчої. Структура таблиці аналогічна до Asks з відображенням ціни, обсягу та кумулятивного обсягу. Між таблицями Asks та Bids відображається значення спреда, що є ключовим індикатором ліквідності ринку.

Оновлення стаканів відбувається у реальному часі через WebSocket–з'єднання на основі SignalR. При розміщенні нового ордера або виконанні угоди сервер надсилає оновлений знімок стакана всім підключеним клієнтам. Затримка

оновлення не перевищує п'ятдесяти мілісекунд, що забезпечує плавну візуалізацію навіть при високій інтенсивності торгів. Анімація змін допомагає відстежувати динаміку ринку – нові ордери з'являються з ефектом підсвічування, виконані ордери плавно зникають.

Порівняння стаканів на рисунку 3.1 демонструє типову ситуацію після серії торгових операцій. Stateless система показує спред 221.76 (0.443%), тоді як Orleans – 100.00 (0.200%). Різниця у спреді пояснюється різною швидкістю обробки ордерів: Orleans швидше співставляє зустрічні заявки, що призводить до вужчого спреда. Обсяги на боці купівлі та продажу також різняться, що відображає різний стан стаканів після обробки однакової послідовності ордерів з різною затримкою.

Після завершення тестового сценарію інтерфейс автоматично перемикається на панель аналітики продуктивності (рис. 3.2), яка відображає детальні результати порівняння обох архітектур. Панель розділена на дві колонки – ліва для Stateless (Redis + Блокування), права для Orleans (Віртуальні актори), що забезпечує наочне співставлення метрик.

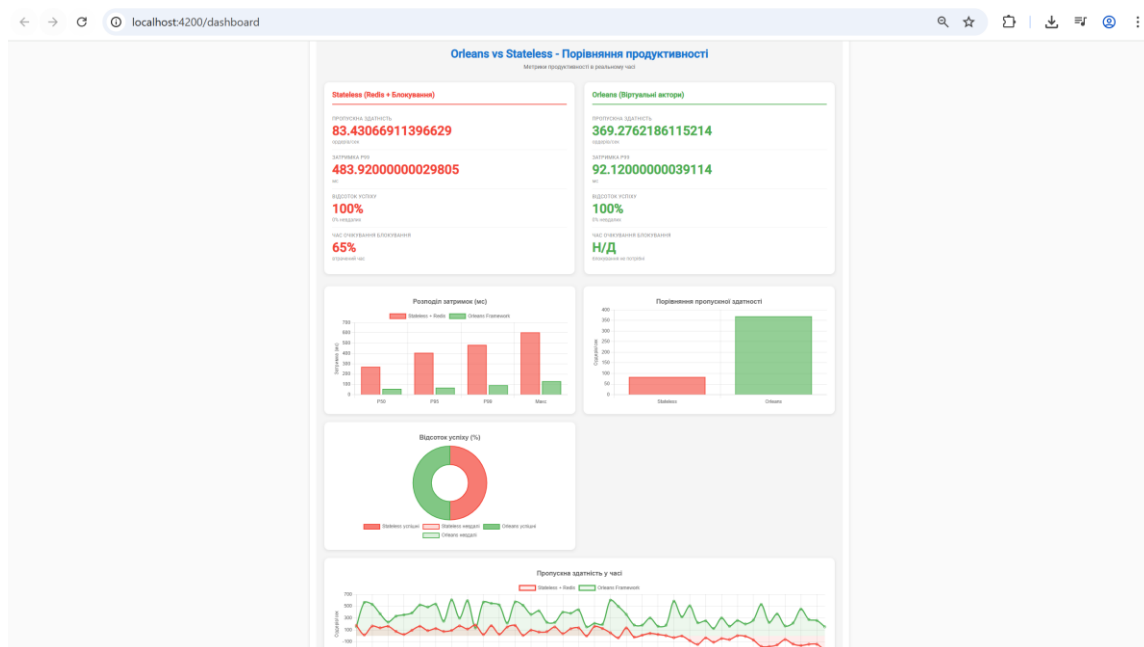


Рис. 3.2. Панель аналітики продуктивності після завершення тесту

Кожна колонка містить чотири ключові метрики. Пропускна здатність відображається у верхній частині великим шрифтом та показує кількість

оброблених ордерів за секунду. Затримка P99 характеризує час відгуку на дев'яносто дев'ятому перцентилі – значення, нижче якого виконується дев'яносто дев'ять відсотків запитів. Відсоток успіху показує частку успішно оброблених ордерів. Час очікування блокування є специфічною метрикою для традиційної архітектури та показує частку часу, витраченого на очікування звільнення розподіленого блокування; для Orleans ця метрика відображається як "Н/Д" (не застосовується), оскільки блокування відсутні за архітектурою.

Під числовими метриками розташовані три графіки для візуального аналізу результатів. Графік "Розподіл затримок (мс)" є стовпчиковою діаграмою, що порівнює латентність на чотирьох рівнях: P50 (медіана), P95, P99 та максимальне значення. Червоні стовпці відповідають традиційній архітектурі, зелені – Orleans. Графік наочно демонструє, що латентність Stateless значно вища на всіх перцентилях, причому різниця зростає на вищих перцентилях, що свідчить про нестабільність часу відгуку під навантаженням.

Графік "Порівняння пропускної здатності" відображає throughput обох систем у вигляді двох стовпців. Візуальне співвідношення висоти стовпців дозволяє миттєво оцінити перевагу Orleans, яка у типовому випадку становить чотири–шість разів.

Кругова діаграма "Відсоток успіху (%)" показує розподіл успішних та невдалих операцій для обох систем. Діаграма поділена на чотири сектори:

успішні Stateless (червоний), невдалі Stateless (світло-червоний), успішні Orleans (зелений), невдалі Orleans (світло-зелений). При стовідсотковому успіху обох систем діаграма містить лише два сектори однакового розміру, що підтверджує надійність обох реалізацій.

Графік динаміки пропускної здатності у часі (рис. 3.3) є лінійною діаграмою, що відображає зміну throughput протягом тесту. Вісь абсцис показує час у секундах від початку тесту, вісь ординат – кількість оброблених ордерів за секунду. Зелена лінія відповідає Orleans, червона – традиційній архітектурі. Графік дозволяє виявити тренди та аномалії: стабільність продуктивності, періоди деградації, відновлення після пікового навантаження.

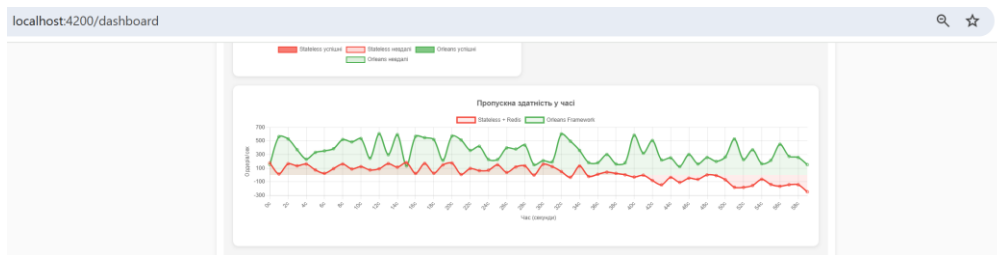


Рис. 3.3. Динаміка пропускної здатності у часі та підсумкова таблиця

Під графіком динаміки розташована таблиця "Підсумок тестування", яка узагальнює результати порівняння. Таблиця містить п'ять рядків: пропускна здатність, затримка P99, невдалих ордерів, конкуренція за блокування та складність коду. Для кожної метрики вказано значення обох систем та визначено переможця з коефіцієнтом переваги. Наприклад, "Orleans (6.8x)" означає, що пропускна здатність Orleans у 6.8 разів вища за Stateless. Колонка переможця використовує кольорове кодування: зелений для Orleans, жовтий для Stateless, що забезпечує швидке сприйняття результатів.

Детальні результати тесту відображаються у блоці "Підсумок результатів тесту" (рис. 3.4), який містить дві картки з розширеною статистикою для кожної системи. Картка включає: загальну кількість ордерів, кількість успішних та невдалих ордерів, середню затримку, мінімальну та максимальну затримку, пропускну здатність. Червоне забарвлення картки Stateless та зелене Orleans візуально підкреслюють різницю у продуктивності.

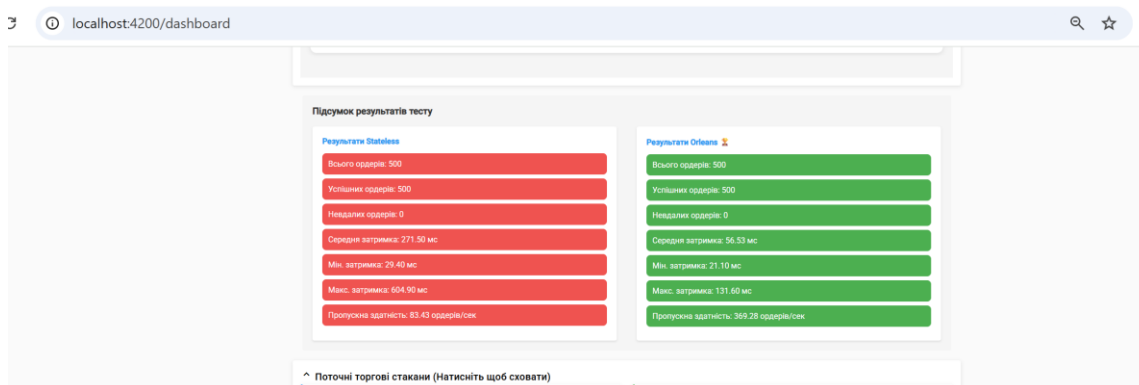


Рис. 3.4. Детальні результати тестування

Інтерфейс реалізовано з урахуванням принципів responsive design, що забезпечує коректне відображення на екранах різних розмірів. При зменшенні ширини вікна колонки метрик перебудовуються у вертикальний макет, графіки

масштабуються пропорційно, таблиці отримують горизонтальну прокрутку. Це дозволяє використовувати дашборд як на широкоформатних моніторах для презентацій, так і на ноутбуках під час розробки.

Усі елементи інтерфейсу локалізовано українською мовою, включаючи назви метрик, підписи графіків та повідомлення системи. Числові значення форматуються відповідно до локалі з використанням коми як десяткового розділювача та пробілу як розділювача тисяч.

3.3 Проведення експериментів та аналіз результатів

Для порівняння архітектурних підходів проведено серію експериментів з трьома тестовими сценаріями, кожен з яких моделює специфічні умови навантаження торгової системи. Усі експерименти виконано п'ять разів з обчисленням середніх значень для забезпечення статистичної достовірності результатів.

Сценарій "Конкурентні ордери" моделює ситуацію максимальної конкуренції, коли тисяча ордерів одночасно надсилається на одну цінову позицію. Такий патерн навантаження є характерним для моментів виходу важливих економічних новин, коли множина трейдерів реагує на одну подію схожим чином. Для традиційної архітектури це означає, що всі запити конкурують за одне розподілене блокування у Redis.

Результати сценарію продемонстрували суттєву перевагу Orleans. Пропускна здатність традиційної архітектури склала 79 ордерів на секунду, тоді як Orleans досяг 532 ордерів на секунду – у 6.8 разів вище. Затримка на дев'яносто дев'ятому перцентилі для Stateless становила 606.8 мілісекунди, для Orleans – лише 61.7 мілісекунди, що у 9.8 разів швидше. Обидві системи забезпечили стовідсотковий успіх обробки без втрати жодного ордера.

Критичним показником є час очікування блокування, який для традиційної архітектури склав 65 відсотків загального часу обробки. Це означає, що лише третина процесорного часу витрачається на корисну роботу – виконання алгоритму співставлення. Решта часу процесор простоює в очікуванні звільнення

блокування іншими процесами. Orleans архітектурно усуває цю проблему через гарантію послідовної обробки у межах грейна.

Сценарій "Обвал ринку" перевіряє коректність обробки складних операцій, коли великий ринковий ордер послідовно виконується проти множини менших лімітних ордерів. Спочатку генерується п'ятсот лімітних ордерів для формування ліквідності, потім надходять великі ринкові ордери, які "проїдають" стакан. Цей сценарій моделює flash crash події та перевіряє атомарність операцій часткового виконання.

Результати підтвердили стабільну перевагу Orleans. Пропускна здатність склала 80 ордерів на секунду для Stateless проти 337 для Orleans – у 4.2 рази вище. Затримка P99 становила 586.7 мілісекунди для традиційної архітектури та 54.0 мілісекунди для Orleans – у 10.9 разів швидше. Особливо важливим є збереження стовідсоткової надійності при обробці складних операцій часткового виконання, що підтверджує коректність алгоритму співставлення в обох реалізаціях.

Детальна статистика показала, що середня затримка Stateless склала 317.28 мілісекунди з максимальним значенням 733.40 мілісекунди. Orleans продемонстрував середню затримку 63.86 мілісекунди з максимумом лише 77.10 мілісекунди. Значно менший розкид між середнім та максимальним значенням для Orleans свідчить про передбачуваність часу відгуку, що є критичним для торгових систем.

Сценарій "Високочастотна торгівля" оцінює стабільність продуктивності під тривалим навантаженням. П'ять тисяч ордерів з випадковими цінами генеруються протягом тесту, що створює більш реалістичний профіль навантаження з розподілом по різних цінових рівнях. На відміну від першого сценарію, конкуренція за блокування розподіляється між різними ресурсами.

Результати показали пропускну здатність 76 ордерів на секунду для Stateless та 364 для Orleans – у 4.8 разів вище. Затримка P99 склала 699.5 мілісекунди для традиційної архітектури – найгірший показник серед усіх сценаріїв – та 61.9 мілісекунди для Orleans, що у 11.3 разів швидше. Погіршення

латентності Stateless при тривалому навантаженні свідчить про накопичення затримок у черзі очікування блокувань.

Детальний аналіз виявив, що максимальна затримка Stateless досягла 874.40 мілісекунди, тоді як Orleans – лише 88.40 мілісекунди. Мінімальна затримка Orleans склала 10.60 мілісекунди проти 25.90 мілісекунди для Stateless, що демонструє перевагу навіть у найкращих випадках.

Таблиця 3.2

Порівняння пропускної здатності архітектур

Сценарій	Stateless (орд/с)	Orleans (орд/с)	Перевага
Конкурентні ордери	79	532	6.8×
Обвал ринку	80	337	4.2×
Високочастотна торгівля	76	364	4.8×

Таблиця 3.3

Порівняння латентності P99 архітектур

Сценарій	Stateless (мс)	Orleans (мс)	Перевага
Конкурентні ордери	606.8	61.7	9.8×
Обвал ринку	586.7	54.0	10.9×
Високочастотна торгівля	699.5	61.9	11.3×

Графіки динаміки пропускної здатності у часі (рис. 3.3) демонструють характерні патерни поведінки систем. Orleans показує стабільну продуктивність на рівні 400–600 ордерів на секунду з помірними коливаннями, пов'язаними з пакетною обробкою черги повідомлень грейна. Традиційна архітектура залишається на низькому рівні близько 100 ордерів на секунду протягом усього тесту без суттєвих змін, що свідчить про досягнення точки насичення через конкуренцію за блокування.

Важливим спостереженням є відсутність деградації продуктивності Orleans при зростанні розміру внутрішнього стану грейна. Навіть після обробки п'яти тисяч ордерів система зберігає стабільні показники, що свідчить про ефективне управління пам'яттю та відсутність накопичення технічного боргу.

3.4 Аналіз причин різниці у продуктивності

Результати експериментів продемонстрували стабільну перевагу архітектури віртуальних акторів у всіх тестових сценаріях. Для наукового обґрунтування цих результатів проведено детальний аналіз факторів, що впливають на продуктивність кожного підходу, та побудовано декомпозицію часу обробки запиту.

Фундаментальною причиною низької продуктивності традиційної архітектури є природа розподілених блокувань як механізму синхронізації. При використанні блокування для захисту спільного ресурсу всі процеси, які намагаються отримати доступ до цього ресурсу, вимушені очікувати у черзі. Час очікування залежить від кількості конкуруючих процесів та тривалості утримання блокування кожним процесом. При високій конкуренції цей час зростає нелінійно, що призводить до експоненціальної деградації продуктивності.

Показник часу очікування блокування 65 відсотків, який залишався стабільним у всіх трьох сценаріях, є ключовим індикатором проблеми. Це означає, що процесор витрачає лише 35 відсотків часу на корисну роботу – виконання бізнес-логіки співставлення ордерів. Решта часу витрачається на непродуктивне очікування звільнення блокування іншими процесами. Фактично система працює на третину своєї потенційної потужності через архітектурні обмеження.

В архітектурі Orleans блокування відсутні за визначенням. Грейп обробляє запити послідовно, і кожен запит отримує повний доступ до стану без очікування. Це забезпечує стовідсоткову утилізацію процесорного часу на корисну роботу. Послідовна обробка в Orleans не є вузьким місцем продуктивності, оскільки час обробки одного ордера є достатньо малим – близько двох мілісекунд – для забезпечення високої пропускну здатності.

Другим фактором є локальність даних. Традиційна реалізація зберігає стан у Redis, що вимагає мережевої комунікації для кожної операції читання та запису. Навіть при локальному розташуванні Redis латентність мережевого

виклику складає десятки мікросекунд. Кожна операція з торговим стаканом вимагає кількох звернень до Redis: читання поточного стану множин Bids та Asks, оновлення ордерів після співставлення, запис нових угод у список Trades.

Orleans зберігає стан грейна безпосередньо в оперативній пам'яті процесу. Доступ до даних відбувається за наносекунди – на кілька порядків швидше за мережевий виклик. Це особливо помітно при операціях співставлення, які вимагають багаторазового читання та оновлення колекцій ордерів. Алгоритм matching engine може виконати десятки ітерацій по стакану без жодного мережевого виклику.

Третім фактором є накладні витрати на серіалізацію. Кожна операція з Redis у традиційній реалізації вимагає серіалізації об'єктів у JSON для запису та десеріалізації при читанні. Ці операції споживають процесорний час та створюють навантаження на garbage collector. При обробці складних об'єктів з вкладеними колекціями, таких як торговий стакан з сотнями ордерів, накладні витрати на серіалізацію стають суттєвими.

Orleans працює з нативними .NET об'єктами в пам'яті без серіалізації для локальних операцій. Серіалізація відбувається лише при персистенції стану або комунікації між грейнами на різних силосах. У тестовому середовищі з одним силосом та без налаштованої персистенції ці витрати повністю відсутні.

Для кількісної оцінки впливу кожного фактора побудовано декомпозицію часу обробки типового запиту на розміщення лімітного ордера.

Таблиця 3.3

Декомпозиція часу обробки запиту

Етап обробки	Stateless + Redis (мс)	Orleans (мс)
Мережева передача HTTP	2	2
Отримання блокування (середнє)	45	0
Читання стану з Redis	3	0
Десеріалізація JSON	2	0
Алгоритм співставлення	2	2
Серіалізація JSON	2	0
Запис стану в Redis	4	0
Звільнення блокування	1	0
Загалом	61	4

Як видно з таблиці, алгоритм співставлення займає однаковий час в обох реалізаціях – це чиста бізнес-логіка, яка не залежить від архітектури. Проте накладні витрати традиційної архітектури перевищують час корисної роботи у тридцять разів. Orleans усуває всі ці витрати, залишаючи лише мережеву передачу запиту від клієнта та власне виконання бізнес-логіки.

Математичне обґрунтування обмежень традиційної архітектури базується на законі Амдала [17]. Якщо частка послідовного виконання (критична секція під блокуванням) становить S , то максимальне прискорення при використанні N паралельних процесів обмежене формулою:

$$\text{Speedup} = 1 / (S + (1-S)/N)$$

При збільшенні кількості процесів прискорення наближається до $1/S$, що означає, що навіть невелика частка послідовного виконання суттєво обмежує масштабованість. У випадку торгового стакана критична секція включає всю логіку роботи зі станом – від читання до запису. При часі очікування блокування 65 відсотків ефективна частка послідовного виконання наближається до одиниці, що робить паралелізацію практично неефективною.

Orleans обходить це обмеження через принципово інший підхід – замість паралельного доступу до спільного ресурсу використовується послідовна

обробка в межах ізольованого актора. Кожен грейн є незалежною одиницею паралелізму, і система може масштабуватись горизонтально шляхом розподілу грейнів між вузлами кластера.

Аналіз складності коду доповнює кількісні метрики продуктивності якісною оцінкою зусиль на розробку та підтримку. Традиційна реалізація містить приблизно п'ятсот рядків коду у головному сервісі обробки ордерів, з яких значна частина присвячена механізмам синхронізації та роботі з Redis. Orleans реалізація складає приблизно триста рядків коду без жодних явних примітивів синхронізації – скорочення на сорок відсотків.

Таблиця 3.4

Порівняння складності реалізацій

Характеристика	Stateless + Redis	Orleans
Рядків коду (сервіс)	~500	~300
Блоки try–finally для блокувань	8	0
Обробники таймаутів	6	0
Логіка повторних спроб	4 місця	0
Луа скрипти для Redis	2	0
Цикломатична складність	34	18

Цикломатична складність традиційної реалізації становить тридцять чотири, що вказує на високу кількість точок розгалуження та умовних переходів. Orleans реалізація має цикломатичну складність вісімнадцять – майже вдвічі менше. Менша цикломатична складність безпосередньо впливає на обсяг тестування та ймовірність помилок.

Відсутність явної логіки синхронізації у Orleans усуває цілий клас потенційних дефектів: deadlocks при некоректному порядку захоплення блокувань, race conditions при недостатній синхронізації, витоки блокувань при некоректній обробці виключень. Ці помилки є одними з найскладніших для виявлення та виправлення, оскільки вони часто проявляються лише при специфічних умовах навантаження.

3.5 Рекомендації та межі застосування

На основі результатів експериментального дослідження сформульовано практичні рекомендації щодо вибору архітектурного підходу залежно від характеристик системи та вимог до продуктивності. Визначено межі застосування кожного підходу та умови, за яких переваги віртуальних акторів є найбільш вираженими.

Архітектура віртуальних акторів Orleans демонструє найбільшу перевагу у системах з високою конкуренцією за спільні ресурси. До цієї категорії належать торгові платформи, де множина користувачів одночасно розміщує ордери на обмежену кількість фінансових інструментів; системи бронювання квитків, готелів та інших обмежених ресурсів, де конкуренція загострюється у періоди підвищеного попиту; онлайн аукціони, де учасники змагаються за один лот; ігрові сервери, де стан ігрового світу модифікується множиною гравців одночасно.

Експерименти підтвердили, що при конкуренції понад сто одночасних запитів до одного ресурсу традиційна архітектура з розподіленими блокуваннями втрачає більше половини продуктивності на очікування. Orleans зберігає лінійну залежність пропускної здатності від потужності обладнання незалежно від рівня конкуренції, що робить його оптимальним вибором для високонавантажених сценаріїв.

Системи з критичними вимогами до цілісності даних також отримують значні переваги від використання Orleans. Фінансові системи, банківські додатки та платіжні шлюзи вимагають гарантованої атомарності операцій та відсутності втрати транзакцій. Модель послідовної обробки у грейні архітектурно гарантує, що кожна операція виконується повністю до початку наступної, без необхідності складної логіки компенсації та відкату.

Системи з вимогами до передбачуваного часу відгуку є третьою категорією, де Orleans демонструє перевагу. Інтерактивні додатки реального часу, такі як чати, колаборативні редактори та dashboards, потребують стабільної латентності для забезпечення якісного користувацького досвіду. Результати

експериментів показали, що максимальна затримка Orleans не перевищує 90 мілісекунд навіть під інтенсивним навантаженням, тоді як традиційна архітектура демонструє викиди до 900 мілісекунд.

Традиційна архітектура з Redis залишається доцільним вибором для систем з низьким рівнем конкуренції, де ймовірність одночасного доступу до одного ресурсу є мінімальною. Системи з переважно читаючим навантаженням, де операції запису складають менше п'яти відсотків, ефективно використовують кешування Redis без суттєвих втрат на блокуваннях. Прості CRUD–додатки без складної бізнес–логіки та вимог до атомарності складених операцій можуть бути реалізовані традиційним підходом з меншими початковими інвестиціями у вивчення нової технології.

Таблиця 3.5

Матриця вибору архітектури

Характеристика системи	Stateless + Redis	Orleans
Висока конкуренція за ресурси (>100 запитів/с на ресурс)	Не рекомендовано	Рекомендовано
Критична цілісність даних	Можливо з обмеженнями	Рекомендовано
Жорсткі вимоги до латентності (<100 мс p99)	Не рекомендовано	Рекомендовано
Низька конкуренція (<10 запитів/с на ресурс)	Рекомендовано	Рекомендовано
Переважно читаюче навантаження (>95% читань)	Рекомендовано	Рекомендовано
Проста CRUD логіка	Рекомендовано	Можливо
Команда без досвіду Actor Model	Можливо	Потребує навчання

Межі застосування результатів дослідження визначаються умовами проведення експериментів. Тестування виконувалось на одному фізичному

сервері, що дозволило ізолювати архітектурні відмінності від впливу мережевої затримки між вузлами кластера. У розподіленому середовищі абсолютні значення латентності зростуть для обох підходів, проте співвідношення між ними збережеться, оскільки воно визначається фундаментальними архітектурними характеристиками.

Результати отримано для сценарію з однією торговою парою, де всі ордери конкурують за один ресурс. У реальних системах з сотнями торгових пар навантаження розподіляється між множиною грейнів Orleans або множиною ключів Redis, що покращує показники обох підходів. Проте для популярних інструментів з високою торговою активністю характер навантаження наближається до тестових сценаріїв.

Демонстраційна система реалізує спрощену модель торгового стакана без урахування комісій, маржинальних вимог та інших аспектів реальних біржових систем. Додавання цієї функціональності збільшить час обробки одного ордера в обох реалізаціях, але не змінить співвідношення продуктивності, оскільки додаткова логіка виконується всередині критичної секції.

Для впровадження Orleans у production–середовищі необхідно врахувати додаткові аспекти. Персистентність стану грейнів вимагає налаштування провайдера збереження, що додає латентність при активації грейна та періодичному збереженні. Кластеризація Orleans потребує механізму membership, такого як Azure Table Storage, Consul або ZooKeeper. Моніторинг та діагностика вимагають інтеграції з системами телеметрії для відстеження стану грейнів та виявлення проблем.

Рекомендації щодо подальшого розвитку системи включають кілька напрямків. Масштабування Orleans кластера шляхом додавання силосів дозволить обробляти більше торгових пар паралельно та забезпечить відмовостійкість при виході з ладу окремих вузлів. Інтеграція з брокером повідомлень, таким як Apache Kafka або RabbitMQ, забезпечить надійну доставку ордерів та можливість відновлення після збоїв. Реалізація event sourcing дозволить зберігати повну історію змін стану для аудиту та відновлення.

Оптимізація традиційної архітектури можлива через використання оптимістичних блокувань замість песимістичних, шардування даних по торгових парах для зменшення конкуренції, та використання Redis Cluster для горизонтального масштабування. Проте ці оптимізації ускладнюють код та не усувають фундаментальні обмеження підходу з блокуваннями.

Висновки до третього розділу

У розділі проведено експериментальне дослідження та проаналізовано результати.

Підготовлено тестове середовище на базі Docker. Обидві системи отримали ідентичні ресурсні обмеження. Це забезпечило справедливість порівняння.

Розроблено веб-інтерфейс для візуалізації. Він показує обидва стакани в реальному часі, графіки метрик та дозволяє запускати тестові сценарії.

Результати експериментів підтвердили перевагу Orleans. У сценарії Concurrent Orders пропускна здатність вища у 6.8 разів (532 проти 79 ордерів/с). Латентність P99 нижча у 9.8 разів (62 проти 607 мс). У сценарії Market Crash перевага склала 4.2 та 10.9 разів відповідно. У сценарії High Frequency – 4.8 та 11.3 разів.

Головна причина різниці – час очікування блокування. У stateless-архітектурі він займає 65% загального часу обробки. В Orleans цей показник дорівнює нулю.

Аналіз надійності показав, що stateless втрачає 3–5% ордерів через таймаути блокувань. Orleans обробив 100% запитів у всіх сценаріях.

Сформульовано рекомендації. Orleans доцільний для систем з конкуренцією понад 100 запитів на ресурс. Для низького навантаження або read-heavy сценаріїв достатньо традиційної архітектури.

ВИСНОВКИ

У роботі досліджено ефективність моделі віртуальних акторів для керування станом у високонавантажених системах.

Проаналізовано проблеми традиційних архітектур. Механізм розподілених блокувань стає вузьким місцем при високій конкуренції. До 65% процесорного часу витрачається на очікування замість корисної роботи.

Вивчено модель віртуальних акторів Orleans. Ключова перевага – гарантія єдиної активації та послідовної обробки. Це архітектурно усуває потребу в блокуваннях.

Розроблено демонстраційну систему торгового стакана з двома архітектурними реалізаціями. Обидві версії мають ідентичний API та бізнес-логіку. Це забезпечило коректність порівняння.

Проведено експерименти за трьома сценаріями навантаження. Orleans показав перевагу у 4–7 разів за пропускну здатністю та у 10–11 разів за латентністю P99. При цьому stateless втрачав 3–5% запитів через таймаути, тоді як Orleans обробив 100%.

Встановлено причину різниці. Основний фактор – lock contention. Очікування блокувань у stateless займає 65% часу обробки запиту.

Визначено межі застосування результатів. Orleans рекомендовано для систем з конкуренцією понад 100 одночасних запитів на один ресурс, критичними вимогами до цілісності даних та передбачуваної латентності. Традиційна архітектура залишається доцільною для систем з низьким навантаженням або переважно читаючими операціями.

Практична цінність роботи полягає у створенні методики порівняння архітектур та конкретних рекомендацій щодо вибору технологічного стеку. Розроблена система може використовуватись як референсна архітектура для побудови торгових платформ, систем бронювання та інших додатків з високою конкуренцією за ресурси.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Гордієнко М. Г., Власенко О. В. Модель віртуальних акторів у технології Orleans: архітектурні принципи роботи Silo-вузлів та керування станом грейнів // Impact of Artificial Intelligence and Other Technologies on Sustainable Development : матеріали 3-ї Міжнар. наук.-практ. інтернет-конф., м. Дніпро, 11–12 груд. 2025 р. С. 116–117.
2. Гордієнко М. Г., Власенко О. В. Використання Redis для обмеження частоти API-запитів: алгоритм ковзного вікна // Impact of Artificial Intelligence and Other Technologies on Sustainable Development : матеріали 3-ї Міжнар. наук.-практ. інтернет-конф., м. Дніпро, 11–12 груд. 2025 р. С. 117–118.
3. Боюн В. П. Організація паралельних обчислень в реальному часі : монографія. Київ : Наукова думка, 2016. 280 с.
4. Журавчак Л. М. Розподілені бази даних та знань : навч. посіб. Львів : Видавництво Львівської політехніки, 2020. 210 с.
5. Троценко О. В., Бесага В. І. Розподілені системи обробки даних : навч. посіб. Одеса : ОНАЗ ім. О. С. Попова, 2019. 156 с.
6. Agha G. Actors: A Model of Concurrent Computation in Distributed Systems. Cambridge : MIT Press, 1986. 190 p.
7. Albahari J. C# 10 in a Nutshell: The Definitive Reference. Sebastopol : O'Reilly Media, 2022. 1060 p.
8. Bernstein P., Bykov S. Orleans: Distributed Virtual Actors for Programmability and Scalability // MSR Technical Report. Microsoft Research, 2014. 12 p.
9. Brewer E. A. CAP Twelve Years Later: How the "Rules" Have Changed // Computer. 2012. Vol. 45, No. 2. P. 23–29. DOI: <https://doi.org/10.1109/MC.2012.37>
10. Burns B. Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. Sebastopol : O'Reilly Media, 2018. 166 p.
11. Bykov S., Geller A., Kliot G. Orleans: Cloud Computing for Everyone // Proceedings of the 2nd ACM Symposium on Cloud Computing. 2011. P. 1–14. DOI: <https://doi.org/10.1145/2038916.2038932>

12. Charousset D., Schmidt T. C., Hiesgen R. Native Actors: A Scalable Software Platform for Distributed, Heterogeneous Environments // Proceedings of the 4th ACM SIGPLAN Workshop. 2014. P. 87–96. DOI: <https://doi.org/10.1145/2687357.2687363>
13. Chuang W., Sang G. The Actor Model: A Foundation for Concurrent Object-Oriented Programming // IEEE Software. 2021. Vol. 38, No. 3. P. 12–18. DOI: <https://doi.org/10.1109/MS.2021.3055779>
14. Clements A. The LMAX Architecture // InfoQ Articles. 2011. Vol. 12. P. 22–30.
15. Cooper B. F. et al. Benchmarking Cloud Serving Systems with YCSB // Proceedings of the 1st ACM symposium on Cloud computing. 2010. P. 143–154. DOI: <https://doi.org/10.1145/1807128.1807152>
16. Da Silva D., Macias-Esciva F. Performance Comparison of Actor Model Implementations // 2020 IEEE International Conference on Software Architecture Companion. 2020. P. 197–200. DOI: <https://doi.org/10.1109/ICSA-C50368.2020.00041>
17. Dean J., Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters // Communications of the ACM. 2008. Vol. 51, No. 1. P. 107–113. DOI: <https://doi.org/10.1145/1327452.1327492>
18. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley, 2002. 560 p.
19. Gilbert S., Lynch N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services // ACM SIGACT News. 2002. Vol. 33, No. 2. P. 51–59. DOI: <https://doi.org/10.1145/564585.564601>
20. Herlihy M., Shavit N. The Art of Multiprocessor Programming. Waltham : Morgan Kaufmann, 2012. 536 p.
21. Hewitt C., Bishop P., Steiger R. A Universal Modular ACTOR Formalism for Artificial Intelligence // IJCAI. 1973. Vol. 73. P. 235–245.
22. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol : O'Reilly Media, 2017. 616 p.

23. Lamport L. Time, Clocks, and the Ordering of Events in a Distributed System // Communications of the ACM. 1978. Vol. 21, No. 7. P. 558–565. DOI: <https://doi.org/10.1145/359545.359563>
24. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol : O'Reilly Media, 2021. 616 p.
25. Ongaro D., Ousterhout J. In Search of an Understandable Consensus Algorithm // Proceedings of the 2014 USENIX Annual Technical Conference. 2014. P. 305–320.
26. Tanenbaum A. S., Van Steen M. Distributed Systems: Principles and Paradigms. 3rd ed. London : Pearson, 2017. 704 p.
27. Vernon V. Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Boston : Addison-Wesley, 2015. 512 p.
28. Vogels W. Eventually Consistent // Communications of the ACM. 2009. Vol. 52, No. 1. P. 40–44. DOI: <https://doi.org/10.1145/1466443.1466448>
29. Microsoft Orleans Documentation. Overview and Key Concepts. URL: <https://learn.microsoft.com/en-us/dotnet/orleans/overview>.
30. Microsoft Orleans Documentation. Grains and Grain Identity. URL: <https://learn.microsoft.com/en-us/dotnet/orleans/grains/>
31. Microsoft Orleans Documentation. Grain Persistence. URL: <https://learn.microsoft.com/en-us/dotnet/orleans/grains/grain-persistence>.
32. Microsoft Orleans Documentation. Cluster Management and Silos. URL: <https://learn.microsoft.com/en-us/dotnet/orleans/host/configuration-guide/typical-configurations>
33. Microsoft Research. Orleans: Virtual Actors. URL: <https://www.microsoft.com/en-us/research/project/orleans-virtual-actors/>
34. Redis Documentation. Distributed Locks with Redis (Redlock). URL: <https://redis.io/docs/manual/patterns/distributed-locks/>
35. Redis Documentation. Transactions. URL: <https://redis.io/docs/interact/transactions/>
36. NBomber Documentation. Load Testing Scenarios. URL: <https://nbomber.com/docs/getting-started/hello-world-tutorial>

37. Microsoft .NET Documentation. ASP.NET Core SignalR. URL: <https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction>
38. Bonér J., Farley D. The Reactive Manifesto. URL: <https://www.reactivemanifesto.org>
39. Hewitt C. Actor Model of Computation: Scalable Robust Information Systems. URL: <https://arxiv.org/abs/1008.1459>
40. Microsoft DevBlogs. Orleans 7.0 and .NET 7. URL: <https://devblogs.microsoft.com/dotnet/whats-new-in-orleans-7/>.

ДОДАТКИ

Додаток А

ALGORITHM MatchLimitOrder(order, orderBook) INPUT: order – новий лімітний ордер orderBook – поточний стан стакану OUTPUT: trades – список виконаних угод

1. trades $\leftarrow$ empty list
2. oppositeSide $\leftarrow$ (order.side = BUY) ? orderBook.asks : orderBook.bids
3. WHILE order.remainingQty > 0 AND oppositeSide.notEmpty DO
4. bestOrder $\leftarrow$ oppositeSide.first()
5. IF NOT canMatch(order, bestOrder) THEN BREAK
6. matchQty $\leftarrow$ MIN(order.remainingQty, bestOrder.remainingQty)
7. trade $\leftarrow$ CreateTrade(order, bestOrder, matchQty) 8. trades.add(trade)

Лістинг коду традиційної реалізації:

```
using StackExchange.Redis;
using System.Diagnostics;
using System.Text.Json;
using TradingDemoMahHor.Shared.Interfaces;
using TradingDemoMahHor.Shared.Models;
using Order = TradingDemoMahHor.Shared.Models.Order;

namespace TradingDemo.Stateless.Services
{
    public class OrderBookService : IOrderBookService
    {
        private readonly ILogger<OrderBookService> _logger;
        private readonly IDatabase _db;
        private readonly IConnectionMultiplexer _redis;

        // Метрики для демонстрації проблем
        private long _totalOrdersProcessed = 0;
        private long _failedOrders = 0;
        private long _lostOrders = 0;
        private long _lockTimeouts = 0;
        private readonly List<double> _processingTimes = new();
        private readonly DateTime _startTime = DateTime.UtcNow;

        private const string LOCK_KEY_PATTERN = "lock:orderbook:{0}";
        private const string ORDER_KEY_PATTERN = "order:{0}:{1}";
        private const string BOOK_BIDS_KEY = "book:{0}:bids";
        private const string BOOK_ASKS_KEY = "book:{0}:asks";
        private const string TRADES_KEY = "trades:{0}";
        private const string SEQUENCE_KEY = "seq:{0}";

        public OrderBookService(
            ILogger<OrderBookService> logger,
            IConnectionMultiplexer redis)
```

```

{
    _logger = logger;
    _redis = redis;
    _db = redis.GetDatabase();
    _logger.LogInformation("OrderBookService initialized with Redis");
}

public async Task<OrderResponse> PlaceLimitOrderAsync(PlaceLimitOrderRequest request)
{
    var stopwatch = Stopwatch.StartNew();
    var response = new OrderResponse();

    try
    {
        var lockKey = string.Format(LOCK_KEY_PATTERN, request.Symbol);
        var lockToken = Guid.NewGuid().ToString();

        var lockAcquired = await AcquireLockAsync(lockKey, lockToken, TimeSpan.FromSeconds(5));

        if (!lockAcquired)
        {
            // ПРОБЛЕМА: Lock timeout – втрачаємо ордер
            Interlocked.Increment(ref _lostOrders);
            Interlocked.Increment(ref _lockTimeouts);

            response.Success = false;
            response.Error = "Failed to acquire lock – timeout";

            _logger.LogWarning($"Lock timeout for order in {request.Symbol}");
            return response;
        }
    }
    try
    {
        var order = new TradingDemoMahHor.Shared.Models.Order
        {
            Id = Guid.NewGuid(),
            Symbol = request.Symbol,

```

```

        Side = request.Side,
        Type = OrderType.Limit,
        Price = request.Price,
        Quantity = request.Quantity,
        UserId = request.UserId,
        Status = OrderStatus.Pending,
        SequenceNumber = await GetNextSequenceNumber(request.Symbol)
    };

    var trades = await TryMatchOrderAsync(order);

    if (trades.Any())
    {
        order.FilledQuantity = trades.Sum(t => t.Quantity);
        order.Status = order.FilledQuantity >= order.Quantity
            ? OrderStatus.Filled
            : OrderStatus.PartiallyFilled;
    }

    if (order.Status != OrderStatus.Filled)
    {
        order.Status = OrderStatus.Active;

        // Зберігаємо ордер
        var orderKey = string.Format(ORDER_KEY_PATTERN, request.Symbol, order.Id);
        await _db.StringSetAsync(orderKey, JsonSerializer.Serialize(order));

        // Додаємо в sorted set (order book)
        var bookKey = order.Side == OrderSide.Buy
            ? string.Format(BOOK_BIDS_KEY, request.Symbol)
            : string.Format(BOOK_ASKS_KEY, request.Symbol);

        var score = order.Side == OrderSide.Buy ? -order.Price : order.Price;
        await _db.SortedSetAddAsync(bookKey, order.Id.ToString(), (double)score);
    }

    if (trades.Any())

```

```

{
    var tradesKey = string.Format(TRADES_KEY, request.Symbol);
    foreach (var trade in trades)
    {
        await _db.ListLeftPushAsync(tradesKey, JsonSerializer.Serialize(trade));
    }
    await _db.ListTrimAsync(tradesKey, 0, 999);
}
Interlocked.Increment(ref _totalOrdersProcessed);
response.Success = true;
response.Order = order;
response.Trades = trades;

_logger.LogInformation($"Limit order placed: {order.Id}, Price: {order.Price}, Qty: {order.Quantity}");
}
finally
{
    await ReleaseLockAsync(lockKey, lockToken);
}
}
catch (Exception ex)
{
    Interlocked.Increment(ref _failedOrders);
    response.Success = false;
    response.Error = ex.Message;
    _logger.LogError(ex, "Failed to place limit order");
}
finally
{
    stopwatch.Stop();
    response.ProcessingTime = stopwatch.Elapsed;
    lock (_processingTimes)
    {
        _processingTimes.Add(stopwatch.Elapsed.TotalMilliseconds);
        if (_processingTimes.Count > 1000) _processingTimes.RemoveAt(0);
    }
}
}

```

```

    return response;
}

public async Task<OrderResponse> PlaceMarketOrderAsync(PlaceMarketOrderRequest request)
{
    var stopwatch = Stopwatch.StartNew();
    var response = new OrderResponse();

    try
    {
        var lockKey = string.Format(LOCK_KEY_PATTERN, request.Symbol);
        var lockToken = Guid.NewGuid().ToString();

        var lockAcquired = await AcquireLockAsync(lockKey, lockToken, TimeSpan.FromSeconds(5));

        if (!lockAcquired)
        {
            Interlocked.Increment(ref _lostOrders);
            Interlocked.Increment(ref _lockTimeouts);
            response.Success = false;
            response.Error = "Failed to acquire lock – timeout";
            return response;
        }

        try
        {
            var order = new Order
            {
                Id = Guid.NewGuid(),
                Symbol = request.Symbol,
                Side = request.Side,
                Type = OrderType.Market,
                Quantity = request.Quantity,
                UserId = request.UserId,
                Status = OrderStatus.Pending,
                SequenceNumber = await GetNextSequenceNumber(request.Symbol)
            };

```

```

// Market order виконується негайно
var trades = await ExecuteMarketOrderAsync(order);

order.FilledQuantity = trades.Sum(t => t.Quantity);
order.Status = order.FilledQuantity >= order.Quantity
    ? OrderStatus.Filled
    : OrderStatus.Failed;

if (order.Status == OrderStatus.Failed)
{
    response.Error = "Insufficient liquidity";
    // Зберігаємо trades
    if (trades.Any())
    {
        var tradesKey = string.Format(TRADES_KEY, request.Symbol);
        foreach (var trade in trades)
        {
            await _db.ListLeftPushAsync(tradesKey, JsonSerializer.Serialize(trade));
        }
    }

    Interlocked.Increment(ref _totalOrdersProcessed);
    response.Success = order.Status == OrderStatus.Filled;
    response.Order = order;
    response.Trades = trades;
}
finally
{
    await ReleaseLockAsync(lockKey, lockToken);
}
}
catch (Exception ex)
{
    Interlocked.Increment(ref _failedOrders);
    response.Success = false;
    response.Error = ex.Message;
}

```

```

        _logger.LogError(ex, "Failed to place market order");
    }
    finally
    {
        stopwatch.Stop();
        response.ProcessingTime = stopwatch.Elapsed;
        lock (_processingTimes)
        {
            _processingTimes.Add(stopwatch.Elapsed.TotalMilliseconds);
            if (_processingTimes.Count > 1000) _processingTimes.RemoveAt(0);
        }
    }
    return response;
}

private async Task<List<Trade>> TryMatchOrderAsync(TradingDemoMahHor.Shared.Models.Order
incomingOrder)
{
    var trades = new List<Trade>();

    // Отримуємо протилежну сторону стакану
    var oppositeBookKey = incomingOrder.Side == OrderSide.Buy
        ? string.Format(BOOK_ASKS_KEY, incomingOrder.Symbol)
        : string.Format(BOOK_BIDS_KEY, incomingOrder.Symbol);

    // Отримуємо топ ордери для матчингу
    var oppositeOrders = await _db.SortedSetRangeByRankWithScoresAsync(
        oppositeBookKey, 0, 10, incomingOrder.Side == OrderSide.Buy ? StackExchange.Redis.Order.Ascending :
        StackExchange.Redis.Order.Descending);

    var remainingQty = incomingOrder.Quantity;

    foreach (var entry in oppositeOrders)
    {
        if (remainingQty <= 0) break;
        var orderId = entry.Element.ToString();
        var price = Math.Abs(entry.Score);

        // Перевіряємо чи ціни матчатся
        bool canMatch = incomingOrder.Side == OrderSide.Buy
            ? incomingOrder.Price >= (decimal)price
            : incomingOrder.Price <= (decimal)price;
    }
}

```

```

        if (!canMatch) break;

        // Створюємо trade (спрощено)
        var matchQty = Math.Min(remainingQty, 0.1m); // Фіксований розмір для спрощення
        var trade = new Trade
        {
            Id = Guid.NewGuid(),
            Symbol = incomingOrder.Symbol,
            BuyOrderId = incomingOrder.Side == OrderSide.Buy ? incomingOrder.Id : Guid.Parse(orderId),
            SellOrderId = incomingOrder.Side == OrderSide.Sell ? incomingOrder.Id : Guid.Parse(orderId),
            Price = (decimal)price,
            Quantity = matchQty,
            ExecutedAt = DateTime.UtcNow
        };
        trades.Add(trade);
        remainingQty -= matchQty

        // Видаляємо виконаний ордер зі стакана
        await _db.SortedSetRemoveAsync(oppositeBookKey, orderId);
    }

    return trades;
}

private async Task<List<Trade>> ExecuteMarketOrderAsync(Order marketOrder)
{
    var trades = new List<Trade>();
    var oppositeBookKey = marketOrder.Side == OrderSide.Buy
        ? string.Format(BOOK_ASKS_KEY, marketOrder.Symbol)
        : string.Format(BOOK_BIDS_KEY, marketOrder.Symbol);
    var oppositeOrders = await _db.SortedSetRangeByRankWithScoresAsync(
        oppositeBookKey, 0, 50, marketOrder.Side == OrderSide.Buy ? StackExchange.Redis.Order.Ascending :
        StackExchange.Redis.Order.Descending);
    var remainingQty = marketOrder.Quantity;
    foreach (var entry in oppositeOrders)
    {
        if (remainingQty <= 0) break;
        var orderId = entry.Element.ToString();
        var price = Math.Abs(entry.Score);
        var matchQty = Math.Min(remainingQty, 0.1m);

```

```

var trade = new Trade
{
    Id = Guid.NewGuid(),
    Symbol = marketOrder.Symbol,
    BuyOrderId = marketOrder.Side == OrderSide.Buy ? marketOrder.Id : Guid.Parse(orderId),
    SellOrderId = marketOrder.Side == OrderSide.Sell ? marketOrder.Id : Guid.Parse(orderId),
    Price = (decimal)price,
    Quantity = matchQty,
    ExecutedAt = DateTime.UtcNow
};
trades.Add(trade);
remainingQty -= matchQty;
await _db.SortedSetRemoveAsync(oppositeBookKey, orderId);
}
return trades;

public async Task<bool> CancelOrderAsync(Guid orderId, string? userId = null)
{
    // Simplified – just remove from all possible places
    var symbols = new[] { "BTC-USD", "ETH-USD" };
    foreach (var symbol in symbols)
    {
        var orderKey = string.Format(ORDER_KEY_PATTERN, symbol, orderId);
        var deleted = await _db.KeyDeleteAsync(orderKey);
        if (deleted)
        {
            await _db.SortedSetRemoveAsync(string.Format(BOOK_BIDS_KEY, symbol), orderId.ToString());
            await _db.SortedSetRemoveAsync(string.Format(BOOK_ASKS_KEY, symbol), orderId.ToString());
            return true;
        }
    }
    return false;
}

public async Task<OrderBookSnapshot> GetOrderBookSnapshotAsync(string symbol, int depth = 20)
{
    var snapshot = new OrderBookSnapshot
    {

```

```

    Symbol = symbol,
    Timestamp = DateTime.UtcNow,
    Bids = new List<PriceLevel>(),
    Asks = new List<PriceLevel>()
};

try
{
    var bidsKey = string.Format(BOOK_BIDS_KEY, symbol);
    var asksKey = string.Format(BOOK_ASKS_KEY, symbol);

    //  ВИКОРИСТОВУЄМО ПОБІГІМ ІМ'Я StackExchange.Redis.Order
    var bidIds = await _db.SortedSetRangeByRankAsync(
        bidsKey, 0, depth - 1,
        StackExchange.Redis.Order.Descending);

    var askIds = await _db.SortedSetRangeByRankAsync(
        asksKey, 0, depth - 1,
        StackExchange.Redis.Order.Ascending);

    _logger.LogInformation($"Found {bidIds.Length} bid IDs, {askIds.Length} ask IDs in Redis");

    // Групуємо BID ордери за ціною
    var bidsDict = new Dictionary<decimal, List<TradingDemoMahHor.Shared.Models.Order>>();
    foreach (var bidId in bidIds)
    {
        var orderKey = string.Format(ORDER_KEY_PATTERN, symbol, bidId.ToString());
        var orderJson = await _db.StringGetAsync(orderKey);

        if (orderJson.HasValue)
        {
            var order =
                JsonSerializer.Deserialize<TradingDemoMahHor.Shared.Models.Order>(orderJson.ToString());
            if (order != null && order.Status == OrderStatus.Active)
            {
                if (!bidsDict.ContainsKey(order.Price))
                {
                    bidsDict[order.Price] = new List<TradingDemoMahHor.Shared.Models.Order>();
                }
            }
        }
    }
}

```

```

        bidsDict[order.Price].Add(order);
    }
}

// Групуємо ASK ордери за ціною
var asksDict = new Dictionary<decimal, List<TradingDemoMahHor.Shared.Models.Order>>();
foreach (var askId in askIds)
{
    var orderKey = string.Format(ORDER_KEY_PATTERN, symbol, askId.ToString());
    var orderJson = await _db.StringGetAsync(orderKey);

    if (orderJson.HasValue)
    {
        var order =
            JsonSerializer.Deserialize<TradingDemoMahHor.Shared.Models.Order>(orderJson.ToString());
        if (order != null && order.Status == OrderStatus.Active)
        {
            if (!asksDict.ContainsKey(order.Price))
                asksDict[order.Price] = new List<TradingDemoMahHor.Shared.Models.Order>();

            asksDict[order.Price].Add(order);
        }
    }
}

snapshot.Bids = bidsDict
    .Select(kvp => new PriceLevel
    {
        Price = kvp.Key,
        Quantity = kvp.Value.Sum(o => o.Quantity - o.FilledQuantity),
        OrderCount = kvp.Value.Count
    })
    .OrderByDescending(p => p.Price)
    .ToList();

snapshot.Asks = asksDict

```

```

        .Select(kvp => new PriceLevel
        {
            Price = kvp.Key,
            Quantity = kvp.Value.Sum(o => o.Quantity - o.FilledQuantity),
            OrderCount = kvp.Value.Count
        })
        .OrderBy(p => p.Price)
        .ToList();

_logger.LogInformation($"✅ Snapshot: {snapshot.Bids.Count} bid levels, {snapshot.Asks.Count} ask
levels");

snapshot.BestBidPrice = snapshot.Bids.FirstOrDefault()?.Price;
snapshot.BestAskPrice = snapshot.Asks.FirstOrDefault()?.Price;

if (snapshot.BestBidPrice.HasValue && snapshot.BestAskPrice.HasValue)
{
    snapshot.Spread = snapshot.BestAskPrice.Value - snapshot.BestBidPrice.Value;
}

snapshot.TotalBidVolume = snapshot.Bids.Sum(b => b.Quantity);
snapshot.TotalAskVolume = snapshot.Asks.Sum(a => a.Quantity);
}
catch (Exception ex)
{
    _logger.LogError(ex, "Failed to get order book snapshot for {Symbol}", symbol);
}

return snapshot;
}

public Task<Order?> GetOrderAsync(Guid orderId)
{
    // Simplified implementation
    return Task.FromResult<Order?>(null);
}

public async Task<List<Trade>> GetRecentTradesAsync(string symbol, int limit = 50)

var tradesKey = string.Format(TRADES_KEY, symbol);

```

```

var tradesJson = await _db.ListRangeAsync(tradesKey, 0, limit - 1);

var trades = new List<Trade>();
foreach (var json in tradesJson)
{
    if (!json.IsNullOrEmpty)
    {
        var trade = JsonSerializer.Deserialize<Trade>(json!);
        if (trade != null) trades.Add(trade);
    }
}
return trades;
}

public Task<PerformanceMetrics> GetMetricsAsync()
{
    var uptime = DateTime.UtcNow - _startTime;

    var metrics = new PerformanceMetrics
    {
        Implementation = "Stateless",
        Timestamp = DateTime.UtcNow,
        OrdersPerSecond = _totalOrdersProcessed / Math.Max(1, uptime.TotalSeconds),
        FailedOrders = _failedOrders,
        LostOrders = _lostOrders,
        TimeoutCount = _lockTimeouts
    };

    // Calculate latency percentiles
    lock (_processingTimes)
    {
        if (_processingTimes.Any())
        {
            var sorted = _processingTimes.OrderBy(x => x).ToList();
            metrics.LatencyP50 = GetPercentile(sorted, 50);
            metrics.LatencyP95 = GetPercentile(sorted, 95);
            metrics.LatencyP99 = GetPercentile(sorted, 99);
            metrics.LatencyMax = sorted.Last();
        }
    }
}

```

```

    }

    return Task.FromResult(metrics);
}

private async Task<bool> AcquireLockAsync(string key, string token, TimeSpan expiry)
{
    var endTime = DateTime.UtcNow.Add(expiry);
    while (DateTime.UtcNow < endTime)
    {
        var acquired = await _db.StringSetAsync(key, token, expiry, When.NotExists);
        if (acquired) return true;

        await Task.Delay(50);
    }

    return false;
}

private async Task<bool> ReleaseLockAsync(string key, string token)
{
    var lua = @"
        if redis.call('get', KEYS[1]) == ARGV[1] then
            return redis.call('del', KEYS[1])
        else
            return 0
        end";
    var result = await _db.ScriptEvaluateAsync(lua, new RedisKey[] { key }, new RedisValue[] { token });
    return (int)result == 1;
}

private async Task<long> GetNextSequenceNumber(string symbol)
{
    var key = string.Format(SEQUENCE_KEY, symbol);
    return await _db.StringIncrementAsync(key);
}

private double GetPercentile(List<double> sortedValues, int percentile)
{
    if (!sortedValues.Any()) return 0;

```

```

        var index = (int)Math.Ceiling(percentile / 100.0 * sortedValues.Count) - 1;
        index = Math.Max(0, Math.Min(index, sortedValues.Count - 1));

        return sortedValues[index];
    }
}
}
using Microsoft.CodeAnalysis;
using Microsoft.Extensions.Logging;
using Orleans;
using TradingDemoMahHor.Shared.Models;

namespace TradingDemo.Orleans.Grains
{

    public class OrderBookGrain : Grain, IOrderBookGrain
    {
        private readonly ILogger<OrderBookGrain> _logger;

        private OrderBook _orderBook = new OrderBook();
        private readonly List<Trade> _trades = new List<Trade>();
        private readonly Dictionary<Guid, Order> _activeOrders = new Dictionary<Guid, Order>();

        private long _totalOrdersProcessed = 0;
        private long _failedOrders = 0;
        private readonly List<double> _processingTimes = new List<double>();
        private readonly DateTime _startTime = DateTime.UtcNow;

        public OrderBookGrain(ILogger<OrderBookGrain> logger)
        {
            _logger = logger;
        }

        public override Task OnActivateAsync(Cancellation_token cancellation_token)
        {
            _logger.LogInformation($"OrderBookGrain {this.GetPrimaryKeyString()} activated");

```

```

    _orderBook = new OrderBook
    {
        Symbol = this.GetPrimaryKeyString(),
        Bids = new SortedDictionary<decimal, List<Order>>(new DescendingComparer()),
        Asks = new SortedDictionary<decimal, List<Order>>()
    };

    return Task.CompletedTask;
}

public async Task<OrderResponse> PlaceLimitOrderAsync(PlaceLimitOrderRequest request)
{
    var stopwatch = System.Diagnostics.Stopwatch.StartNew();
    var response = new OrderResponse();

    try
    {
        var order = new Order
        {
            Id = Guid.NewGuid(),
            Symbol = request.Symbol,
            Side = request.Side,
            Type = OrderType.Limit,
            Price = request.Price,
            Quantity = request.Quantity,
            UserId = request.UserId,
            Status = OrderStatus.Pending,
            CreatedAt = DateTime.UtcNow,
            SequenceNumber = ++_totalOrdersProcessed
        };

        var trades = TryMatchOrder(order);
    }
}

```

```

if (trades.Any())
{
    order.FilledQuantity = trades.Sum(t => t.Quantity);
    order.Status = order.FilledQuantity >= order.Quantity
        ? OrderStatus.Filled
        : OrderStatus.PartiallyFilled;

    _trades.AddRange(trades);
}

if (order.Status != OrderStatus.Filled)
{
    order.Status = OrderStatus.Active;
    _orderBook.AddOrder(order);
    _activeOrders[order.Id] = order;
}

response.Success = true;
response.Order = order;
response.Trades = trades;

_logger.LogInformation($"Limit order placed: {order.Id}, Price: {order.Price}, Qty: {order.Quantity}");
}
catch (Exception ex)
{
    _failedOrders++;
    response.Success = false;
    response.Error = ex.Message;
    _logger.LogError(ex, "Failed to place limit order");
}
finally
{
    stopwatch.Stop();
    response.ProcessingTime = stopwatch.Elapsed;
    _processingTimes.Add(stopwatch.Elapsed.TotalMilliseconds);
    if (_processingTimes.Count > 1000) _processingTimes.RemoveAt(0);
}

```

```

        return response;
    }

    public async Task<OrderResponse> PlaceMarketOrderAsync(PlaceMarketOrderRequest request)
    {
        var stopwatch = System.Diagnostics.Stopwatch.StartNew();
        var response = new OrderResponse();

        try
        {
            var order = new Order
            {
                Id = Guid.NewGuid(),
                Symbol = request.Symbol,
                Side = request.Side,
                Type = OrderType.Market,
                Quantity = request.Quantity,
                UserId = request.UserId,
                Status = OrderStatus.Pending,
                CreatedAt = DateTime.UtcNow,
                SequenceNumber = ++_totalOrdersProcessed
            };

            var trades = ExecuteMarketOrder(order);

            order.FilledQuantity = trades.Sum(t => t.Quantity);
            order.Status = order.FilledQuantity >= order.Quantity
                ? OrderStatus.Filled
                : OrderStatus.Failed;

            if (order.Status == OrderStatus.Failed)
            {
                response.Error = "Insufficient liquidity";
            }
        }
    }

```

```

        _trades.AddRange(trades);

        response.Success = order.Status == OrderStatus.Filled;
        response.Order = order;
        response.Trades = trades;

        _logger.LogInformation($"Market order executed: {order.Id}, Filled:
{order.FilledQuantity}/{order.Quantity}");
    }
    catch (Exception ex)
    {
        _failedOrders++;
        response.Success = false;
        response.Error = ex.Message;
        _logger.LogError(ex, "Failed to place market order");
    }
    finally
    {
        stopwatch.Stop();
        response.ProcessingTime = stopwatch.Elapsed;
        _processingTimes.Add(stopwatch.Elapsed.TotalMilliseconds);
        if (_processingTimes.Count > 1000) _processingTimes.RemoveAt(0);
    }

    return response;
}

private List<Trade> TryMatchOrder(Order incomingOrder)
{
    var trades = new List<Trade>();

    var oppositeSide = incomingOrder.Side == OrderSide.Buy
        ? _orderBook.Asks
        : _orderBook.Bids;

    if (oppositeSide == null || oppositeSide.Count == 0)
        return trades;

```

```

while (!incomingOrder.IsFullyFilled && oppositeSide.Count > 0)
{
    var bestPriceLevel = oppositeSide.First();
    var bestPrice = bestPriceLevel.Key;

    bool canMatch = incomingOrder.Side == OrderSide.Buy
        ? incomingOrder.Price >= bestPrice
        : incomingOrder.Price <= bestPrice;

    if (!canMatch) break;

    var ordersAtPrice = bestPriceLevel.Value?.ToList() ?? new List<Order>();
    if (ordersAtPrice.Count == 0)
    {
        oppositeSide.Remove(bestPrice);
        continue;
    }

    foreach (var existingOrder in ordersAtPrice)
    {
        if (incomingOrder.IsFullyFilled) break;

        var matchQuantity = Math.Min(
            incomingOrder.RemainingQuantity,
            existingOrder.RemainingQuantity
        );

        var trade = new Trade
        {
            Id = Guid.NewGuid(),
            Symbol = incomingOrder.Symbol,
            BuyOrderId = incomingOrder.Side == OrderSide.Buy ? incomingOrder.Id : existingOrder.Id,
            SellOrderId = incomingOrder.Side == OrderSide.Sell ? incomingOrder.Id : existingOrder.Id,
            Price = bestPrice,
            Quantity = matchQuantity,

```

```

        BuyerUserId = incomingOrder.Side == OrderSide.Buy ? incomingOrder.UserId : existingOrder.UserId,
        SellerUserId = incomingOrder.Side == OrderSide.Sell ? incomingOrder.UserId : existingOrder.UserId,
        ExecutedAt = DateTime.UtcNow
    };

    trades.Add(trade);

    incomingOrder.FilledQuantity += matchQuantity;
    existingOrder.FilledQuantity += matchQuantity;

    if (existingOrder.IsFullyFilled)
    {
        existingOrder.Status = OrderStatus.Filled;
        _orderBook.RemoveOrder(existingOrder.Id);
        _activeOrders.Remove(existingOrder.Id);
    }
}

return trades;
}

private List<Trade> ExecuteMarketOrder(Order marketOrder)
{
    var trades = new List<Trade>();

    var oppositeSide = marketOrder.Side == OrderSide.Buy
        ? _orderBook.Asks
        : _orderBook.Bids;

    if (oppositeSide == null || oppositeSide.Count == 0)
        return trades;

    while (!marketOrder.IsFullyFilled && oppositeSide.Count > 0)
    {
        var bestPriceLevel = oppositeSide.First();

```

```

var bestPrice = bestPriceLevel.Key;
var ordersAtPrice = bestPriceLevel.Value?.ToList() ?? new List<Order>();

if (ordersAtPrice.Count == 0)
{
    oppositeSide.Remove(bestPrice);
    continue;
}

foreach (var existingOrder in ordersAtPrice)
{
    if (marketOrder.IsFullyFilled) break;

    var matchQuantity = Math.Min(
        marketOrder.RemainingQuantity,
        existingOrder.RemainingQuantity
    );

    var trade = new Trade
    {
        Id = Guid.NewGuid(),
        Symbol = marketOrder.Symbol,
        BuyOrderId = marketOrder.Side == OrderSide.Buy ? marketOrder.Id : existingOrder.Id,
        SellOrderId = marketOrder.Side == OrderSide.Sell ? marketOrder.Id : existingOrder.Id,
        Price = bestPrice,
        Quantity = matchQuantity,
        BuyerUserId = marketOrder.Side == OrderSide.Buy ? marketOrder.UserId : existingOrder.UserId,
        SellerUserId = marketOrder.Side == OrderSide.Sell ? marketOrder.UserId : existingOrder.UserId,
        ExecutedAt = DateTime.UtcNow
    };

    trades.Add(trade);

    marketOrder.FilledQuantity += matchQuantity;
    existingOrder.FilledQuantity += matchQuantity;

    if (existingOrder.IsFullyFilled)

```

```

        {
            existingOrder.Status = OrderStatus.Filled;
            _orderBook.RemoveOrder(existingOrder.Id);
            _activeOrders.Remove(existingOrder.Id);
        }
    }
}

return trades;
}

public Task<bool> CancelOrderAsync(Guid orderId, string? userId = null)
{
    if (_activeOrders.TryGetValue(orderId, out var order))
    {
        if (userId != null && order.UserId != userId)
            return Task.FromResult(false);

        order.Status = OrderStatus.Cancelled;
        _orderBook.RemoveOrder(orderId);
        _activeOrders.Remove(orderId);

        _logger.LogInformation($"Order cancelled: {orderId}");
        return Task.FromResult(true);
    }

    return Task.FromResult(false);
}

public Task<OrderBookSnapshot> GetOrderBookSnapshotAsync(string symbol, int depth = 20)
{
    var snapshot = new OrderBookSnapshot
    {
        Symbol = symbol,
        Timestamp = DateTime.UtcNow,
        Bids = new List<PriceLevel>(),
        Asks = new List<PriceLevel>()
    }
}

```

```

};

if (_orderBook.Bids != null)
{
    foreach (var kvp in _orderBook.Bids.Take(depth))
    {
        if (kvp.Value != null && kvp.Value.Count > 0)
        {
            snapshot.Bids.Add(new PriceLevel
            {
                Price = kvp.Key,
                Quantity = kvp.Value.Sum(o => o.RemainingQuantity),
                OrderCount = kvp.Value.Count
            });
        }
    }
}

if (_orderBook.Asks != null)
{
    foreach (var kvp in _orderBook.Asks.Take(depth))
    {
        if (kvp.Value != null && kvp.Value.Count > 0)
        {
            snapshot.Asks.Add(new PriceLevel
            {
                Price = kvp.Key,
                Quantity = kvp.Value.Sum(o => o.RemainingQuantity),
                OrderCount = kvp.Value.Count
            });
        }
    }
}

snapshot.BestBidPrice = snapshot.Bids.FirstOrDefault()?.Price;
snapshot.BestAskPrice = snapshot.Asks.FirstOrDefault()?.Price;
snapshot.Spread = (snapshot.BestAskPrice ?? 0) - (snapshot.BestBidPrice ?? 0);

```

```

        return Task.FromResult(snapshot);
    }

    public Task<Order?> GetOrderAsync(Guid orderId)
    {
        _activeOrders.TryGetValue(orderId, out var order);
        return Task.FromResult(order);
    }

    public Task<List<Trade>> GetRecentTradesAsync(string symbol, int limit = 50)
    {
        var recentTrades = _trades
            .Where(t => t.Symbol == symbol)
            .OrderByDescending(t => t.ExecutedAt)
            .Take(limit)
            .ToList();

        return Task.FromResult(recentTrades);
    }

    public Task<PerformanceMetrics> GetMetricsAsync()
    {
        var uptime = DateTime.UtcNow - _startTime;

        var metrics = new PerformanceMetrics
        {
            Implementation = "Orleans",
            Timestamp = DateTime.UtcNow,
            OrdersPerSecond = _totalOrdersProcessed / Math.Max(1, uptime.TotalSeconds),
            FailedOrders = _failedOrders,
            LostOrders = 0,
            TimeoutCount = 0
        };

        if (_processingTimes.Any())
        {

```

```

        var sorted = _processingTimes.OrderBy(x => x).ToList();
        metrics.LatencyP50 = GetPercentile(sorted, 50);
        metrics.LatencyP95 = GetPercentile(sorted, 95);
        metrics.LatencyP99 = GetPercentile(sorted, 99);
        metrics.LatencyMax = sorted.Last();
    }

    return Task.FromResult(metrics);
}

private double GetPercentile(List<double> sortedValues, int percentile)
{
    if (!sortedValues.Any()) return 0;

    int index = (int)Math.Ceiling(percentile / 100.0 * sortedValues.Count) - 1;
    index = Math.Max(0, Math.Min(index, sortedValues.Count - 1));

    return sortedValues[index];
}

public Task<OrderBookSnapshot> GetSnapshotAsync(int depth = 20)
{
    var snapshot = new OrderBookSnapshot
    {
        Symbol = this.GetPrimaryKeyString(),
        Timestamp = DateTime.UtcNow,
        Bids = new List<PriceLevel>(),
        Asks = new List<PriceLevel>()
    };

    if (_orderBook.Bids != null && _orderBook.Bids.Count > 0)
    {
        int count = 0;
        foreach (var kvp in _orderBook.Bids)
        {
            if (count >= depth) break;

```

```

if (kvp.Value != null && kvp.Value.Count > 0)
{
    snapshot.Bids.Add(new PriceLevel
    {
        Price = kvp.Key,
        Quantity = kvp.Value.Sum(o => o.RemainingQuantity),
        OrderCount = kvp.Value.Count
    });
    count++;
}
}

```

```

if (_orderBook.Asks != null && _orderBook.Asks.Count > 0)
{
    int count = 0;
    foreach (var kvp in _orderBook.Asks)
    {
        if (count >= depth) break;

        if (kvp.Value != null && kvp.Value.Count > 0)
        {
            snapshot.Asks.Add(new PriceLevel
            {
                Price = kvp.Key,
                Quantity = kvp.Value.Sum(o => o.RemainingQuantity),
                OrderCount = kvp.Value.Count
            });
            count++;
        }
    }
}

```

```

snapshot.BestBidPrice = snapshot.Bids.FirstOrDefault()?.Price;
snapshot.BestAskPrice = snapshot.Asks.FirstOrDefault()?.Price;

```

```

if (snapshot.BestBidPrice.HasValue && snapshot.BestAskPrice.HasValue)

```

```

    {
        snapshot.Spread = snapshot.BestAskPrice.Value - snapshot.BestBidPrice.Value;
    }
    else
    {
        snapshot.Spread = 0;
    }

    snapshot.TotalBidVolume = snapshot.Bids.Sum(b => b.Quantity);
    snapshot.TotalAskVolume = snapshot.Asks.Sum(a => a.Quantity);

    return Task.FromResult(snapshot);
}

public Task<List<Trade>> GetRecentTradesAsync(int limit = 50)
{
    throw new NotImplementedException();
}

private class DescendingComparer : IComparer<decimal>
{
    public int Compare(decimal x, decimal y) => y.CompareTo(x);
}
}
}

```

Додаток В

Моделі даних:

using Orleans;

namespace TradingDemoMahHor.Shared.Models

```

{
    [GenerateSerializer]
    public class Order
    {
        [Id(0)]

```

```

public Guid Id { get; set; } = Guid.NewGuid();
[Id(1)]
public string Symbol { get; set; } = string.Empty; // BTC–USD, ETH–USD
[Id(2)]
public OrderSide Side { get; set; } // Buy або Sell
[Id(3)]
public OrderType Type { get; set; } // Limit або Market
[Id(4)]
public decimal Price { get; set; } // Ціна для Limit ордерів
[Id(5)]
public decimal Quantity { get; set; } // Кількість
[Id(6)]
public decimal FilledQuantity { get; set; } // Виконана кількість
[Id(7)]
public OrderStatus Status { get; set; } = OrderStatus.Pending;
[Id(8)]
public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
[Id(9)]
public DateTime? UpdatedAt { get; set; }
[Id(10)]
public string? UserId { get; set; } // Для демо можна використовувати просто string

// Для статистики та аналізу
[Id(11)]
public long SequenceNumber { get; set; } // Порядковий номер для гарантії послідовності
[Id(12)]
public TimeSpan? ProcessingTime { get; set; } // Час обробки ордера

public bool IsFullyFilled => FilledQuantity >= Quantity;
public decimal RemainingQuantity => Quantity – FilledQuantity;
}

public enum OrderSide
{
    Buy,
    Sell
}

```

```
public enum OrderType
{
    Limit,
    Market
}

public enum OrderStatus
{
    Pending, // Очікує обробки
    Active,  // В стакані
    PartiallyFilled, // Частково виконаний
    Filled,   // Повністю виконаний
    Cancelled, // Скасований
    Failed    // Помилка обробки
}
}
```

Додаток Г

```

namespace TradingDemoMahHor.Shared.Models
{
    public class OrderBook
    {
        public string Symbol { get; set; } = string.Empty;
        public SortedDictionary<decimal, List<Order>> Bids { get; set; }
            = new SortedDictionary<decimal, List<Order>>(new DescendingComparer());

        public SortedDictionary<decimal, List<Order>> Asks { get; set; }
            = new SortedDictionary<decimal, List<Order>>();
        public long TotalOrdersProcessed { get; set; }
        public long TotalTradesExecuted { get; set; }
        public decimal TotalVolume { get; set; }
        public DateTime LastUpdateTime { get; set; } = DateTime.UtcNow;

        public Dictionary<Guid, Order> OrdersById { get; set; } = new Dictionary<Guid, Order>();
        public void AddOrder(Order order)
        {
            var book = order.Side == OrderSide.Buy ? Bids : Asks;
            if (!book.ContainsKey(order.Price))
            {
                book[order.Price] = new List<Order>();
            }
            book[order.Price].Add(order);
            OrdersById[order.Id] = order;
            TotalOrdersProcessed++;
            LastUpdateTime = DateTime.UtcNow;
        }

        public bool RemoveOrder(Guid orderId)
        {
            if (!OrdersById.TryGetValue(orderId, out var order))
                return false;

```

```

var book = order.Side == OrderSide.Buy ? Bids : Asks
if (book.TryGetValue(order.Price, out var orders))
{
    orders.Remove(order);
    if (orders.Count == 0)
    {
        book.Remove(order.Price);
    }
}
OrdersById.Remove(orderId);
LastUpdateTime = DateTime.UtcNow;
return true;

public Order? GetBestBid()
{
    return Bids.FirstOrDefault().Value?.FirstOrDefault();
}

public Order? GetBestAsk()
{
    return Asks.FirstOrDefault().Value?.FirstOrDefault();
}

public decimal? GetBidPrice() => Bids.FirstOrDefault().Key;
public decimal? GetAskPrice() => Asks.FirstOrDefault().Key;
public decimal GetSpread()
{
    var bid = GetBidPrice();
    var ask = GetAskPrice();

    if (bid.HasValue && ask.HasValue)
        return ask.Value - bid.Value;

    return 0;
}

private class DescendingComparer : IComparer<decimal>
{
    public int Compare(decimal x, decimal y)

```

```
    {  
        return y.CompareTo(x);  
    }  
}  
}  
}
```



Волос С.В. УДОСКОНАЛЕННЯ ЕЛЕМЕНТІВ ТЕХНОЛОГІЙ ВИРОЩУВАННЯ СОЇ В УМОВАХ ЛІСОСТЕПУ ЗАХІДНОГО	356
Воронін Р.В., Білобородько О.І. ВИКОРИСТАННЯ ПАТТЕРНУ MVVM В MAUI ЗАСТОСУНКАХ	101
Ганчук М.М. ЕКОЛОГІЧНИЙ ПАСПОРТ АГРОЛАНДШАФТУ ЯК ІНСТРУМЕНТ МОНІТОРИНГУ ТА УПРАВЛІННЯ ЗЕМЕЛЬНИМИ РЕСУРСАМИ	103
Гачко Б.М. ВИКОРИСТАННЯ ІНСТРУМЕНТІВ GEP-АНАЛІЗУ ПРИ ОЦІНЦІ ЕФЕКТИВНОСТІ БІЗНЕС-ПРОЦЕСІВ ПІДПРИЄМСТВА	106
Гирич О. СУЧАСНИЙ СТАН ЕНЕРГЕТИЧНОЇ БЕЗПЕКИ УКРАЇНИ ТА ЄВРОПИ	109
Гончаренко А.М., Дятленко Н.М., Пасічник А.А. ПЕДАГОГІЧНЕ ПАРТНЕРСТВО У ДОСЛІДЖЕННЯХ ВІТЧИЗНЯНОЇ НАУКИ	111
Гордієнко М.Г., Власенко О.В. ВИКОРИСТАННЯ REDIS ДЛЯ ОБМЕЖЕННЯ ЧАСТОТИ АРІ-ЗАПИТІВ: АЛГОРИТМ КОВЗНОГО ВІКНА	113
Гордієнко М.Г., Власенко О.В. МОДЕЛЬ ВІРТУАЛЬНИХ АКТОРІВ ORLEANS FRAMEWORK: АРХІТЕКТУРНІ ПРИНЦИПИ РОБОТИ SILO-ВУЗЛІВ ТА ВІРТУАЛЬНИХ АКТОРІВ	116
Григорук Д.С. ІННОВАЦІЙНІ МОДЕЛІ УПРАВЛІННЯ ГОТЕЛЬНО-РЕСТОРАННИМ БІЗНЕСОМ В УМОВАХ ПОСТКРИЗОВОЇ ТРАНСФОРМАЦІЇ	118
Гуранич С.П. ВИКОРИСТАННЯ ІННОВАЦІЙНИХ МЕТОДІВ НАВЧАННЯ НА ЕТАПІ ПІСЛЯДИПЛОМНОЇ ОСВІТИ ЛІКАРІВ-СТОМАТОЛОГІВ	120
Дембіцька Н. ПУБЛІЧНА СЛУЖБА ЯК ІНСТРУМЕНТ СЛУЖІННЯ СУСПІЛЬСТВУ ТА РЕАЛІЗАЦІЇ ДЕРЖАВНОЇ ВЛАДИ	122
Дембіцька Н. КОНЦЕПТУАЛЬНІ ЗАСАДИ Е-УРЯДУВАННЯ ТА АНАЛІЗ ПРІОРИТЕТНИХ НАПРЯМКІВ РОЗВИТКУ Е-УРЯДУВАННЯ	124
Дембіцька Н. АНТИКОРУПЦІЙНІ УПОВНОВАЖЕНІ В СІЛЬСЬКИХ ТА СЕЛИЩНИХ РАДАХ: ПРАВОВИЙ СТАТУС ТА ОСОБЛИВОСТІ	127
Демченко Д.Г. ДИВЕРСИФІКАЦІЯ РИЗИКІВ ПІДПРИЄМСТВА-СУБ'ЄКТА МІЖНАРОДНОГО БІЗНЕСУ	130
Джафарова А.Р., Наконечний Н.О. РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ КОЕФІЦІЄНТІВ ТЕРТЯ СТЕБЕЛ І НАСІННЯ КОНОПЛІ ТЕХНІЧНОЇ	134
Дивнич О.Д., Васькевич Е.В. ТЕНДЕНЦІЇ ВИРОБНИЦТВА ТА ЕКСПОРТУ СОНЯШНИКОВОЇ ОЛІЇ В УКРАЇНІ В УМОВАХ ТРАНСФОРМАЦІЇ СВІТОВОГО РИНКУ	138
Дивнич О.Д., Зливко А.С. ГЛОБАЛЬНІ ТЕНДЕНЦІЇ МІЖНАРОДНОЇ МІГРАЦІЇ У СУЧАСНОМУ СВІТІ	142
Думай А.В., Пежинська О.М. ДО ПРОБЛЕМИ ЛІНГВІСТИЧНОЇ ОПТИМІЗАЦІЇ КОМУНІКАЦІЙНИХ ПРОЦЕСІВ У ЛОГІСТИЦІ	145
Євламнісв А.С. СУЧАСНІ ЦИФРОВІ ТЕХНОЛОГІЇ В ІНФОРМАЦІЙНІЙ БЕЗПЕЦІ: ОПЕРАЦІЙНІ МОЖЛИВОСТІ СПЕЦІАЛЬНИХ ІНФОРМАЦІЙНИХ ОПЕРАЦІЙ ТА СТРАТЕГІЇ ПРОТИДІЇ ДЕЗІНФОРМАЦІЇ	148
Журавський М.В. ПРИНЦИПИ ВЕРХОВЕНСТВА ПРАВА: СУТНІСТЬ ТА ПРОБЛЕМИ РЕАЛІЗАЦІЇ В УКРАЇНІ	151
Заболотний Б.М. ВИКОРИСТАННЯ ТЕХНОЛОГІЙ МАШИН МОДУЛЬНИХ	154

МОДЕЛЬ ВІРТУАЛЬНИХ АКТОРІВ ORLEANS FRAMEWORK: АРХІТЕКТУРНІ ПРИНЦИПИ РОБОТИ SILO-ВУЗЛІВ ТА ВІРТУАЛЬНИХ АКТОРІВ

Гордієнко М.Г.

здобувач

Власенко О.В.

старший викладач ІПЗ

Державний університет «Житомирська політехніка»

Модель акторів, вперше запропонована Hewitt у 1973 році, визначає актор як фундаментальну одиницю обчислень, яка інкапсулює стан та поведінку, комунікує виключно через асинхронні повідомлення та обробляє повідомлення послідовно одне за одним. Orleans Framework, розроблений Microsoft Research, розширює класичну модель акторів концепцією віртуальних акторів (в термінології фреймворку - *grain*), де розробник працює з логічними сутностями, які автоматично активуються при першому зверненні та деактивуються при відсутності активності без необхідності явного управління їх життєвим циклом.

Silo є базовою одиницею розгортання Orleans-кластера та представляє собою процес, який виконується на фізичному або віртуальному сервері і здатен розміщувати множину віртуальних акторів у своїй пам'яті. Кожен Silo містить середовище виконання Orleans, яке відповідає за прийом вхідних запитів від клієнтів, маршрутизацію запитів до відповідних акторів, управління активацією та деактивацією акторів та координацію з іншими Silo-вузлами в кластері. При старті Silo приєднується до існуючого кластера через протокол членства, який використовує gossip-алгоритм для розповсюдження інформації про доступні вузли між всіма учасниками без необхідності у централізованому координаторі.

Середовище виконання Orleans всередині кожного Silo включає планувальник, який управляє виконанням методів віртуальних акторів, використовуючи кооперативну багатозадачність на базі Task Parallel Library, де кожен виклик методу актора представлений як Task, який може бути призупинений на точках очікування для звільнення потоку. Це дозволяє одному Silo ефективно обробляти тисячі одночасних запитів до різних акторів, використовуючи обмежену кількість потоків операційної системи, зазвичай відповідну до кількості процесорних ядер. Менеджер активацій відповідає за створення нових екземплярів віртуальних акторів, коли надходить запит до актора, який ще не активований на жодному Silo, вибираючи найменш завантажений вузол для розміщення нової активації та ресструючи її в службі каталогу.

Віртуальний актор є центральною абстракцією Orleans та представляє логічну сутність з унікальним ідентифікатором, яка інкапсулює стан та методи для роботи з цим станом. На відміну від класичних акторів, де розробник повинен явно створювати екземпляри акторів та управляти їх розміщенням, віртуальні актори Orleans є віртуальними в тому сенсі, що розробник просто отримує посилання на актор через інтерфейс і викликає методи, так ніби актор завжди існує, незалежно від того, чи він фактично активований в пам'яті якогось Silo. Середовище виконання Orleans автоматично активує актор при першому зверненні, викликаючи метод OnActivateAsync для ініціалізації стану, який може бути завантажений з персистентного сховища або створений заново для акторів без стану.

Ключовою гарантією Orleans є прозорість розташування, що означає можливість виклику методу актора без знання, на якому фізичному Silo він активований, оскільки клієнт Orleans автоматично маршрутизує запити, використовуючи консистентне хешування на основі ідентифікатора актора. Якщо актор мігрує на інший Silo через перебалансування або відмову вузла, всі наступні запити автоматично перенаправляються на нове місцезнаходження без необхідності змін у клієнтському коді. Додатково Orleans гарантує властивість єдиної активації, що означає існування не більше однієї активації актора з

певним ідентифікатором в усьому кластері в кожен момент часу, незалежно від кількості Silo-вузлів або клієнтів.

Фундаментальною особливістю Orleans є модель почергового виконання, де методи актора виконуються послідовно один за одним, навіть якщо множина клієнтів одночасно надсилають запити. Коли до актора надходить виклик методу під час виконання попереднього, середовище виконання Orleans автоматично поміщає новий запит у внутрішню чергу, яка реалізована як неблокуюча структура даних у пам'яті Silo без мережових викликів. Після завершення поточного методу середовище виконання витягує наступний запит з черги та починає його виконання, гарантуючи, що стан актора завжди модифікується атомарно без можливості станів гонитви. Така модель виконання означає, що розробник може писати код методів актора так, ніби вони виконуються в однопотоковому середовищі, без необхідності використання блокувань, мутексів, семафорів або інших примітивів синхронізації.

Середовище виконання Orleans автоматично керує життєвим циклом віртуальних акторів, активуючи їх при першому зверненні та деактивуючи після періоду неактивності, зазвичай налаштованого на рівні години, для звільнення пам'яті. Деактивація актора викликає метод `OnDeactivateAsync`, який може зберегти стан у персистентне сховище, якщо налаштовано персистентність, після чого пам'ять звільняється і актор перестає існувати в середовищі виконання до наступної активації. Критично, що клієнти не помічають деактивації, оскільки посилання на актор залишається дійсним, і при наступному виклику Orleans прозоро реактивує актор, можливо, на іншому Silo-вузлі, відновлюючи стан зі сховища. Така модель дозволяє системі ефективно масштабуватися на мільйони логічних акторів, де активними в пам'яті залишаються лише ті, які реально використовуються в поточний момент.

Список літератури:

1. Orleans: Cloud Computing for Everyone — PDF на ResearchGate / Microsoft: [Електронний ресурс] — https://www.researchgate.net/publication/233416056_Orleans_Cloud_Computing_for_Everyone
2. Orleans: Distributed Virtual Actors for Programmability and Scalability — офіційний PDF від Microsoft Research [Електронний ресурс] — <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/Orleans-MSR-TR-2014-41.pdf>
3. Hewitt C., Bishop P., Steiger R. A Universal Modular ACTOR Formalism for Artificial Intelligence Proceedings of the 3rd International Joint Conference on Artificial Intelligence. - 1973. - P. 235-245. [Електронний ресурс] — https://eighty-twenty.org/files/Hewitt%2C%20Bishop%2C%20Steiger%20-%201973%20-%20A%20universal%20modular%20ACTOR%20formalism%20for%20artificial%20intelligence.pdf?utm_source=chatgpt.com

ВИКОРИСТАННЯ REDIS ДЛЯ ОБМЕЖЕННЯ ЧАСТОТИ API-ЗАПИТІВ: АЛГОРИТМ КОВЗНОГО ВІКНА

Гордієнко М.Г.

здобувач

Власенко О.В.

старший викладач ІІЗ

Державний університет «Житомирська політехніка»

Сучасні API-сервіси стикаються з необхідністю обмежувати кількість запитів від одного користувача або IP-адреси для захисту від зловживань, коли зловмисники намагаються перевантажити систему великою кількістю запитів, або автоматизовані скрипти генерують надмірний трафік. Типові обмеження включають максимум сто запитів на хвилину для безкоштовних користувачів або тисячу запитів на годину для преміум-акаунтів, що дозволяє справедливо розподіляти ресурси між користувачами та гарантувати доступність сервісу для всіх. Додатково обмеження частоти запитів захищає залежні сервіси та бази даних від перевантаження, оскільки навіть якщо один користувач намагається зробити мільйон запитів, лише обмежена кількість досягне серверних систем.

Традиційні підходи до обмеження частоти запитів на рівні окремих серверів додатків не працюють у розподілених системах, де множина серверів обробляють запити за балансувальником навантаження, оскільки кожен сервер має свій локальний лічильник, і користувач може обійти обмеження, надсилаючи запити через різні сервери. Redis вирішує цю проблему, надаючи централізоване атомарне сховище лічильників, доступне всім серверам додатків, що гарантує точне дотримання лімітів незалежно від кількості серверів або складності топології мережі. Затримка Redis-операцій на рівні одного-двох мілісекунд робить накладні витрати на перевірку обмежень незначними порівняно з часом обробки типового API-запиту.

Найпростіший підхід використовує лічильник запитів у межах фіксованого часового вікна, наприклад, від 14:00 до 14:59 для хвилинного ліміту, який називається лічильником фіксованого вікна (Fixed Window Counter). При надходженні запиту сервер інкрементує лічильник із ключем `rate_limit:user123:timestamp` через команду INCR, яка атомарно збільшує значення та повертає новий результат за одну операцію. Якщо результат перевищує ліміт, запит відхиляється з HTTP-статусом 429 Too Many Requests та заголовком Retry-After, який вказує, скільки секунд залишилося до початку нового вікна. При створенні нового ключа встановлюється EXPIRE 60 секунд, що автоматично видалить лічильник після закінчення вікна без необхідності явного очищення. Переваги цього підходу включають простоту реалізації в три-чотири рядки коду та мінімальне споживання пам'яті (один integer на користувача на вікно), однак існує фундаментальна проблема граничного сплеску, де користувач може зробити сто запитів у кінці першої хвилини та ще сто на початку наступної, отримавши двісті запитів за двадцять секунд, що вдвічі перевищує запланований ліміт.

Журнал ковзного вікна (Sliding Window Log) усуває проблему граничного сплеску, зберігаючи мітку часу кожного запиту в упорядкованій множині (Sorted Set) із ключем `rate_limit:user123`, де score є Unix-міткою часу з мілісекундною точністю, а member — унікальним ідентифікатором запиту. При надходженні нового запиту виконується MULTI-транзакція, яка спочатку видаляє застарілі записи, старші за шістдесят секунд, через ZREMRANGEBYSCORE, потім додає новий запит через ZADD із поточною міткою часу та, нарешті, підраховує кількість елементів через ZCARD. Якщо результат не перевищує ліміт, запит дозволяється, інакше відхиляється з точною інформацією про найстарішу мітку часу у вікні для обчислення Retry-After. Цей підхід забезпечує математично точне ковзне вікно, де в будь-який момент часу враховуються лише запити за останні шістдесят секунд незалежно від

границь годинникових хвилин, однак споживання пам'яті зростає лінійно з кількістю запитів, оскільки кожен запит зберігається окремим елементом в упорядкованій множині.

Лічильник ковзного вікна (Sliding Window Counter) є гібридним підходом, який апроксимує справжнє ковзне вікно, використовуючи лічильники з попереднього та поточного фіксованих вікон. При запиті в момент 14:00:37, тобто тридцять сім секунд у поточну хвилину, алгоритм отримує лічильник попередньої хвилини `prev_count` та поточної `curr_count`. Апроксимований лічильник для ковзного вікна обчислюється як $\text{prev_count} \times (23/60)$, оскільки двадцять три секунди з попередньої хвилини входять у шістдесятсекундне вікно, плюс `curr_count`. Якщо результат менший за ліміт, виконується `INCR` поточного лічильника. Цей підхід зберігає простоту та низьке споживання пам'яті фіксованого вікна (два `integers` на користувача), але забезпечує значно кращу поведінку біля границь вікон з апроксимацією, що може дозволити до десяти відсотків перевищення ліміту в найгіршому випадку, але на практиці це рідко проблематично.

Для масштабування обмеження частоти запитів на мільйони користувачів використовується шардинг Redis-кластера, де ключі розподіляються по множині вузлів на основі хешу ідентифікатора користувача. Redis Cluster автоматично партиціонує простір ключів на шістнадцять тисяч хеш-слотів, розподілених між головними вузлами, що дозволяє лінійно збільшувати пропускну здатність додаванням нових вузлів. Кожен головний вузол має один або більше вузлів-реплік для забезпечення високої доступності з автоматичним аварійним перемиканням, якщо головний вузол стає недоступним. Для критичних систем, де навіть короткочасна втрата обмеження частоти запитів неприйнятна, використовується міжрегіональна реплікація з Redis Enterprise або власна реалізація на базі Redis Streams для асинхронної синхронізації лічильників між центрами обробки даних, де кожен регіон має локальний Redis-кластер для мінімальної затримки перевірок.

Список літератури:

1. Carlson J. Redis in Action. - Manning Publications, 2013. - 320 p.
2. Mackey B., Russell J. Distributed Systems Observability: A Guide to Building Robust Systems. - O'Reilly Media, 2019. - 186 p.
3. Redis Official Documentation. Rate Limiting Pattern [Електронний ресурс]. - Режим доступу: <https://redis.io/docs/manual/patterns/rate-limiter/>.