НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету (Директор ННІ)	**Завідувач кафедри**
Інформаційних технологій (назва факультету(ННІ))	Комп'ютерних систем, мереж та кібербезпеки (назва кафедри)
_____ Болбот І.М., д.т.н, проф. (підпис) (ПІБ, вчене звання і ступінь)	_____ Касаткін Д.Ю., к. пед.н., доц. (підпис) (ПІБ, вчене звання і ступінь)
«___»_____2025 р.	«___»_____2025 р.

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Дослідження та вдосконалення розподіленої системи ігрового сервера»

Спеціальність 123 «Комп'ютерна інженерія»
(код і найменування)

Освітня програма Комп'ютерні системи та мережі
(назва)

Орієнтація освітньої програми Освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Гарант освітньої програми

д.т.н., доцент	_____	Шкарупило В. В.
(науковий ступінь та вчене звання)	(підпис)	(ПІБ)

Керівник магістерської кваліфікаційної роботи

доцент	_____	Назаренко В. А.
(науковий ступінь та вчене звання)	(підпис)	(ПІБ)

Виконав _____ Волошин М. Є.
(підпис) (ПІБ)

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних систем, мереж та кібербезпеки
к.пед.н., доц. _____ Касаткін Д.Ю.
(вчене звання і ступінь) (підпис) (ПІБ)
«__» _____ 20__ р.

З А В Д А Н Н Я

ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ ЗДОБУВАЧУ

Волошину Максиму Євгенійовичу
(прізвище, ім'я, по батькові)

Спеціальність 123 «Комп'ютерна інженерія»
(код і найменування)

Освітня програма Комп'ютерні системи та мережі
(назва)

Орієнтація освітньої програми Освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Тема магістерської кваліфікаційної роботи: «Дослідження та вдосконалення розподіленої системи ігрового сервера»

затверджена наказом ректора НУБіП України від "29" жовтня 2024р. № 1941 «С»

Термін подання завершеної роботи на кафедру 14 листопада 2025 р.

Вихідні дані до магістерської кваліфікаційної роботи
Наукові статті та огляди стану предметної області (state-of-the-art) з теми розподілених ігрових серверів, мікросервісних архітектур та хмарного геймінгу.
Перелік питань, що підлягають дослідженню:

1. Проаналізувати теоретичні основи та існуючі архітектурні моделі розподілених ігрових серверів, зокрема для жанру roguelike.
2. Розробити та реалізувати два програмні прототипи серверної системи (на Node.js та C#/.NET) для обробки ігрових сесій.
3. Обґрунтувати шляхи оптимізації та масштабування розробленої системи на основі отриманих емпіричних даних.

Перелік графічного матеріалу (за потреби) _____

Дата видачі завдання " 29 **"** жовтня 2024 р.

Керівник магістерської кваліфікаційної роботи _____ Назаренко В.А.
(підпис) (прізвище та ініціали)

Завдання прийняв до виконання _____ Волошин М.Є.
(підпис) (прізвище та ініціали)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	Аналіз предметної області	29.10.2024 – 15.01.2025	Виконано
2	Теоретичне моделювання підсистем	16.01.2025 – 31.03.2025	Виконано
3	Проведення порівняльного аналізу	01.04.2025 – 31.05.2025	Виконано
4	Оформлення пояснювальної записки	01.06.2025 – 15.10.2025	Виконано
5	Оформлення графічного матеріалу	16.10.2025 – 31.10.2025	Виконано
6	Фіналізація роботи та підготовка до захисту	01.11.2025 – 14.11.2025	Виконано

Студент _____ **В.Ю. Вернигора**

(підпис) (ініціали та прізвище)

Керівник проекту (роботи) _____ **.А.Назаренко**

(підпис) (ініціали та прізвище)

Реферат

Магістерська кваліфікаційна робота: 72 стор., 12 рис., 2 табл., 35 посилань.

Об’єкт дослідження: процес проєктування, розробки та розгортання серверної інфраструктури для сучасних багатокористувацьких онлайн-ігор.

Предмет дослідження: архітектурні моделі, технологічні стеки та методи оптимізації продуктивності розподілених ігрових серверів, зокрема для специфічних вимог ігор жанру roguelike.

Мета роботи: всебічне дослідження архітектур розподілених систем, адаптованих під специфічні вимоги багатокористувацьких ігор жанру roguelike, та подальше вдосконалення обраного архітектурного підходу. Для досягнення цієї мети було поставлено задачі: проаналізувати існуючі технологічні стеки; спроєктувати та реалізувати прототип серверної системи на базі однопотокової платформи Node.js; розробити функціонально ідентичний прототип на базі багатопотокової платформи C#/.NET для проведення порівняльного аналізу; створити методику та провести експериментальне тестування обох систем під симульованим навантаженням. Ключовим завданням є не лише вимірювання показників продуктивності, таких як час відгуку та навантаження на ЦП, але й глибокий аналіз отриманих даних для виявлення та емпіричного обґрунтування фундаментальних "вузьких місць" (CPU-bound проти I/O-bound) кожної архітектури та визначення оптимальних шляхів їх масштабування.

Ключові слова: розподілена система, ігровий сервер, roguelike, Node.js, C#/.NET, аналіз продуктивності, WebSocket.

ЗМІСТ

ВСТУП

Онлайн ігри стали невід'ємною частиною сучасної культури та індустрії розваг. Розвиток технологій та збільшення доступності інтернету призвели до величезного прогресу у цій галузі. У міру розвитку можливостей _технологій розробники знаходять нові способи включити в ігри глибшу соціальну взаємодію та елементи кооперативного геймплею, які залучають широку аудиторію гравців. Ігрова індустрія зазнала кардинальних змін у зв'язку з зростаючою інтеграцією багатокористувацької динаміки у онлайн-ігри. При цьому постійний розвиток індустрії ставить нові вимоги до технологій обробки та передачі даних, оскільки гравці чікують від ігор абсолютної стабільності та низької затримки.

З розвитком технологій і збільшенням числа гравців традиційна клієнт-серверна архітектура стикнулася з низкою проблем, таких як: затримка, синхронізація даних, безпека, масштабованість і в умовах сучасних навантажень демонструє суттєві обмеження, створюючи «вузькі місця» та ускладнюючи масштабування. Саме тому питання побудови надійних та масштабованих розподілених систем ігрових серверів стає одним із ключових у сфері розробки. На противагу їм, сучасні розподілені системи, які базуються на мікросервісах та контейнеризації і пропонують значно вищий рівень відмовостійкості й адаптивності.

Дослідження розподіленої системи ігрового сервера було проведено на грі жанру roguelike (наприклад, Risk of Rain 2). Цей жанр заслуговує окремої уваги, адже характеризується процедурною генерацією рівнів та величезною кількістю незалежних, короткотривалих ігрових сесій. Це ставить унікальні вимоги до серверної частини: здатність миттєво створювати унікальний контент та обслуговувати тисячі паралельних, ізольованих ігор.

Таким чином, виникає актуальна науково-практична задача: обрати та обґрунтувати оптимальний технологічний стек для обробки саме такого, специфічного типу навантаження. Сучасні платформи, такі як Node.js

(однопотокова модель) та C#/.NET (багатопотокова модель), пропонують фундаментально різні підходи до вирішення цієї задачі.

Ця магістерська робота присвячена всебічному дослідженню та вдосконаленню архітектури розподіленого ігрового сервера, обраного для вирішення цих задач. У ході роботи було проаналізовано теоретичні основи, досліджено існуючі платформи та спроєктовано гнучку архітектуру на базі Node.js, орієнтовану на ізоляцію сесій за допомогою Docker.

Ключовою практичною частиною роботи стала реалізація двох функціонально ідентичних серверних прототипів - на Node.js та C#/.NET. Для їх порівняння була розроблена методика та проведена серія ідентичних навантажувальних тестів. Це дозволило провести глибокий порівняльний аналіз отриманих емпіричних даних (часу відгуку та навантаження на ЦП), виявити реальні «вузькі місця» кожної архітектури (CPU-bound ліміт для Node.js та I/O-bound ліміт для .NET) та обґрунтувати шляхи їх оптимізації.

1. ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ РОЗПОДІЛЕНИХ ІГРОВИХ СЕРВЕРІВ

1.1. Поняття та класифікація розподілених систем

Розподілені системи посідають особливе місце у сучасних інформаційних технологіях. Вони являють собою сукупність автономних обчислювальних ресурсів, які взаємодіють між собою через мережу для досягнення спільної мети [1]. Головною особливістю таких систем є те, що для кінцевого користувача вони виглядають як єдине ціле, хоча фактично складаються з багатьох взаємопов'язаних компонентів.

Класичне визначення розподіленої системи наводять Таненбаум і Ван Стеен, які зазначають, що це «сукупність незалежних комп'ютерів, які з позиції користувача функціонують як єдина система». Це означає, що користувач не бачить відмінностей між окремими вузлами, а взаємодіє з системою в цілому.

Основні характеристики розподілених систем:

- Прозорість - користувачеві не потрібно знати, на якому сервері зберігаються його дані чи виконується процес.
- Масштабованість - можливість збільшення обчислювальних ресурсів без радикальних змін у архітектурі.
- Відмовостійкість - здатність продовжувати роботу навіть при виході з ладу окремих вузлів.
- Паралельність - одночасне виконання великої кількості завдань.
- Гнучкість - можливість адаптації до змін у навантаженні або конфігурації.

За архітектурними ознаками розподілені системи можна класифікувати так:

1. Клієнт–серверні системи - найбільш поширені, де клієнт надсилає запити, а сервер відповідає.
2. Peer-to-Peer (P2P) - рівноправна взаємодія вузлів без централізованого сервера.

3. Кластерні системи - група серверів, які працюють як єдиний обчислювальний ресурс.
4. Хмарні системи - інфраструктура, що динамічно розподіляє ресурси між користувачами через інтернет.
5. Мікросервісні архітектури - сучасний підхід, де система складається з набору незалежних сервісів, кожен з яких виконує свою функцію [2].

Для сфери розробки ігор архітектурний підхід розподілених систем набуває воістину вирішального значення.

Саме такі системи надають інструментарій для оперування мільйонними аудиторіями; вони ж є запорукою мінімальної затримки сигналу. Здатність гнучко адаптуватися до раптових сплесків активності - тобто, пікових навантажень - та зберігати непохитну стабільність інфраструктури, нівелюючи відмови окремих серверних вузлів, - все це лежить на їхньому фундаменті.

Ця релевантність стає особливо гострою, коли ми аналізуємо такі жанри, як roguelike, або, власне, будь-які мультиплеєрні проєкти. Причина криється у специфіці: там генерується колосальна кількість незалежних ігрових сесій. І кожна з них висуває, на перший погляд, суперечливу вимогу: бути повністю ізольованою, залишаючись при цьому частиною загальної координованої роботи серверів.

1.2. Архітектурні моделі ігрових серверів

Збереження стану світу. Синхронізація дій гравців. Обробка потоку подій у реальному часі та оркестрація взаємодії клієнтів - увесь цей комплекс завдань покладено на ігровий сервер.

Він і є тим самим центральним елементом мережевої інфраструктури онлайн-гри.

Саме тому вибір архітектури для серверної частини стає фундаментальним рішенням. Воно напряму диктує майбутню продуктивність гри. Воно визначає її

межі масштабованості, загальну надійність і, зрештою, економічну доцільність її підтримки.

Практика ігрової індустрії вже викристалізувала декілька основних архітектурних моделей. Для їхнього предметного порівняння і варто розглянути концептуальну схему (рисунок 1.1).

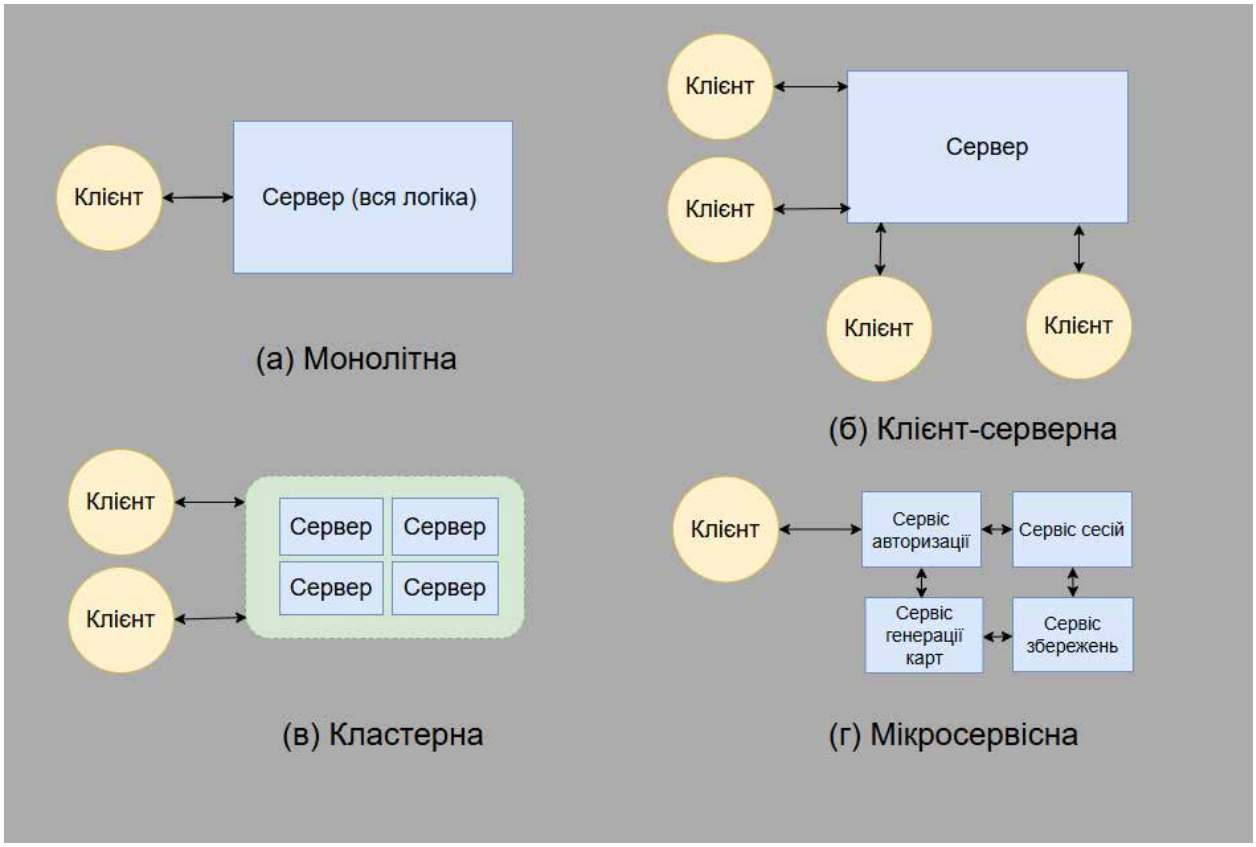

Рисунок 1.1 – Порівняння архітектурних моделей ігрових серверів: а) монолітна; б) клієнт-серверна; в) кластерна; г) мікросервісна.

1. Монолітна архітектура

Монолітна модель. Її ідея - абсолютна концентрація.

Весь без винятку серверний код зосереджений в одному-єдиному програмному блоці. Ігрова логіка, мережеві протоколи, бази даних і системи аналітики - все це реалізовано в рамках єдиного процесу, що виконується на одному сервері.

Такий підхід має очевидні переваги. Це, звісно, простота розробки на початковому етапі. Це мінімальні затримки у взаємодії між компонентами,

оскільки вони знаходяться в єдиному адресному просторі. І, як наслідок, низькі вимоги до інфраструктури.

Проте, недоліки цієї моделі є суттєвими. Перша проблема: погана масштабованість. Обробити тисячі сесій одночасно стає надзвичайно складно. Друга, проблема: залежність від єдиного вузла. Його відмова - це не збій. Його відмова призводить до повної зупинки всієї гри. І, зрештою, наростаюча складність у підтримці та внесенні будь-яких змін.

З огляду на ці фактори, монолітна архітектура, хоч і зустрічалася у старих багатокористувацьких іграх, сьогодні стрімко втрачає свою актуальність.

2. Клієнт–серверна архітектура

Це канонічна модель.

Тут клієнт - чи то ігрова програма на ПК, чи на консолі - відповідає виключно за відображення та введення. Уся без винятку логіка та вся синхронізація - це відповідальність сервера.

Саме сервер зберігає те, що прийнято називати «єдиним джерелом правди» про стан ігрового світу. Клієнт лише формує запити та відображає отримані дані.

Такий чіткий розподіл дає незаперечні переваги:

- Абсолютний централізований контроль над грою.
- Надійний захист від шахрайства, адже клієнт позбавлений права приймати будь-які рішення.
- Висока передбачуваність усієї структури.

Але, звісно, існують і недоліки:

- Обмежена масштабованість, що стає гострою проблемою при зростанні кількості гравців.
- Критично високе навантаження на один центральний сервер, що створює "вузьке місце".
- Гостра необхідність в оптимізації мережевих протоколів.

Попри ці виклики, підхід і досі активно застосовується. Він залишається стандартом для більшості масових онлайн-ігор, особливо в MMORPG [3].

3. Кластерна архітектура

Кластер - це група серверів, що працюють узгоджено, розподіляючи між собою навантаження.

У такій моделі відбувається спеціалізація: окремі вузли беруть на себе відповідальність за конкретні функції. Один - за обробку ігрової логіки. Інший - за роботу з базами даних. Ще один - за балансування запитів чи генерацію контенту.

Переваги такого підходу є вирішальними:

- Висока відмовостійкість. Вихід з ладу одного вузла більше не означає зупинку всієї системи.
- Гнучке масштабування.
- Можливість чіткого розподілу ролей між серверами.

Недоліком, в свою чергу, є зростаюча складність:

- Значне ускладнення процесів налаштування та адміністрування.
- Критична потреба у надійних механізмах балансування навантаження.
- Неминуче зростання витрат на інфраструктуру.

Попри це, саме кластерні рішення демонструють свою максимальну ефективність. Особливо в іграх, де інфраструктура повинна оперувати тисячами одночасних сесій або підтримувати величезні, безшовні відкриті світи.

4. Мікросервісна архітектура [4]

Це вже сучасний підхід, що базується на фундаментальному принципі - декомпозиції. Уся серверна логіка цілеспрямовано розділяється на сукупність дрібних, повністю незалежних сервісів.

Один сервіс відповідає лише за генерацію карт. Інший - лише за обробку боїв. Ще один - лише за систему збережень. А вся комунікація між ними відбувається виключно через API.

Переваги такого підходу:

- Абсолютна модульність і, як наслідок, гнучкість.

- Критично важлива можливість незалежно оновлювати й масштабувати окремі сервіси. Можна посилити один компонент, не впливаючи на решту системи.
- Простота інтеграції з хмарними платформами, для яких технології Docker та Kubernetes є нативними. Docker та Kubernetes - це виглядає як особливо ефективне архітектурне рішення.

Недоліки:

- Значно складніша система комунікації між самими сервісами.
- Пряме зростання вимог до мережевої інфраструктури, адже мережевих взаємодій стає набагато більше.
- гостра необхідність у кваліфікованій DevOps-підтримці.

Незважаючи на це, саме мікросервіси виглядають найбільш перспективним напрямом. Особливо для сучасних багатокористувацьких ігор. А для такого жанру, як roguelike, де вимагається радикальна ізоляція сесій та миттєва генерація контенту, - це майже ідеальне архітектурне рішення.

Таким чином, еволюція архітектур ігрових серверів пройшла чіткий шлях: від простих монолітних моделей до складних кластерних і, зрештою, гнучких мікросервісних систем. Подальший розвиток цього напряму нерозривно пов’язаний із хмарними технологіями, контейнеризацією та автоматизованими підходами до балансування навантаження.

1.3. Проблеми масштабованості та стійкості в багатокористувацьких іграх

Забезпечення стабільної роботи серверної інфраструктури, коли кількість одночасних користувачів стрімко зростає є однією з головних технічних складностей у сфері багатокористувацьких ігор.

Класичні клієнт-серверні моделі чудово працюють для невеликих проєктів. Але коли навантаження сягає тисяч, чи навіть мільйонів підключень, вони миттєво вичерпують свої ресурси. Наслідки цього неминучі: це призводить до

затримок у роботі, суттєвого падіння продуктивності і, як результат, зниження якості ігрового процесу. Саме ця проблема масштабованості - тобто, гостра необхідність обслуговувати величезну кількість гравців одночасно, не втрачаючи швидкодії, - і виходить на перший план [5].

Для вирішення цієї проблеми існують два фундаментальні підходи: вертикальне та горизонтальне масштабування.

Вертикальне, або Scale Up (Scale Up) передбачає нарощування потужності одного-єдиного сервера. Але цей підхід дорогий і, що важливіше, він має свої фізичні обмеження.

Натомість, архітектура сучасних систем повинна мати можливість гнучкого горизонтального масштабування (Scale Out), тобто додавання нових серверних вузлів [6], що є однією з найкращих сучасних практик масштабування багатокористувацьких серверів [7]. Принципову різницю між цими двома підходами ілюструє рисунок 1.2.

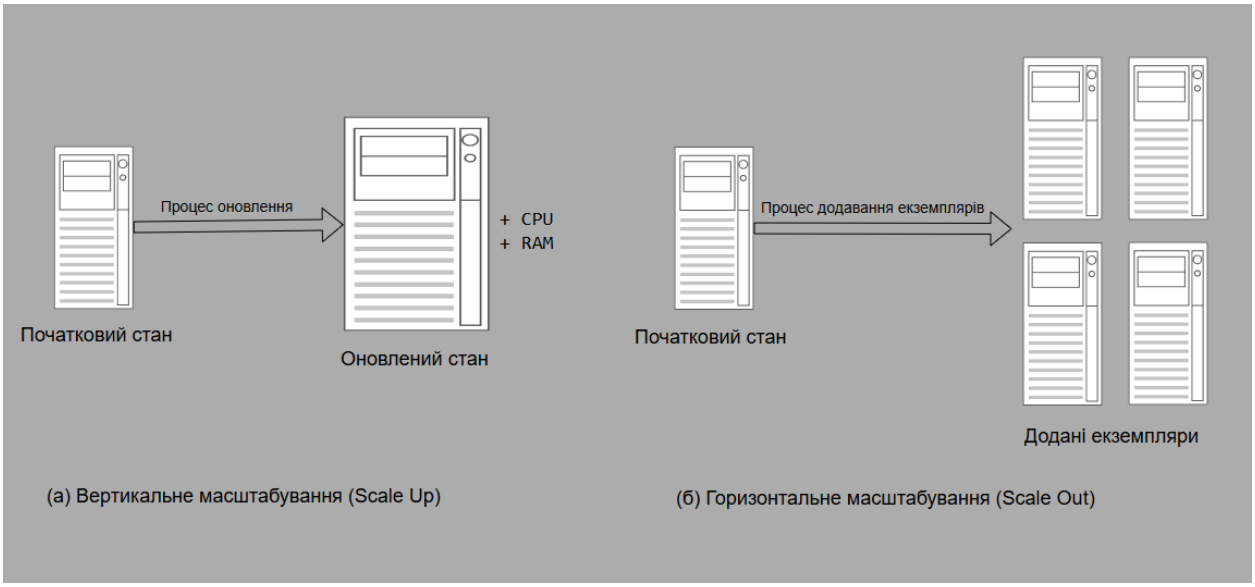

Рисунок 1.2 – Схематичне порівняння вертикального та горизонтального масштабування

Питання стійкості стоїть на одному рівні з масштабованістю.

У великих багатокористувацьких проєктах відмова навіть одного сервера - це може призвести до серйозних наслідків, таких як втрата прогресу гравців, їхнє масове відключення чи повне зупинення ігрового процесу.

Щоб уникнути цього, сучасні системи використовують цілий набір рішень: реплікацію даних, продумане балансування навантаження та механізми автоматичного відновлення збоїв.

Однак реалізація таких рішень завжди пов’язана з додатковими витратами. І, що важливіше, з критичним ускладненням управління всією інфраструктурою.

Прикладом є масові багатокористувацькі ігри, такі як World of Warcraft чи Fortnite, де в мережі одночасно перебувають мільйони користувачів [8].

Для підтримки такої високої стабільності розробники змушені застосовувати складні кластерні системи, які розподіляють гравців між різними вузлами й синхронізують стан світу в реальному часі. Така архітектура, безперечно, знижує ризик збоїв. Але водночас вона породжує нові, не менш складні труднощі, зокрема у забезпеченні консистентності даних.

Отже, проблеми масштабованості та стійкості залишаються ключовими викликами у створенні багатокористувацьких ігор. Їхнє вирішення вимагає впровадження сучасних архітектурних підходів - тих, що здатні поєднати гнучкість, продуктивність і надійність.

Це особливо актуально для ігор у жанрі roguelike, оскільки існує гостра потреба в одночасному обслуговуванні величезної кількості ізольованих ігрових сесій.

1.4. Огляд мережевих технологій для онлайн-ігор

Мережеві технології - це фундамент функціонування багатокористувацьких ігор, адже саме вони визначають швидкість обміну даними, рівень затримки та стабільність сесії. Вибір протоколу залежить від жанру, очікуваної кількості гравців і головного компромісу: що важливіше - точність чи швидкість передачі інформації.

Найбільш традиційний варіант - це TCP (Transmission Control Protocol). TCP забезпечує гарантовану доставку пакетів і суворий контроль їхнього порядку [9]. Його перевага – надійність, проте вона досягається ціною вищої

затримки. Вона неминуче виникає через складні механізми підтвердження доставки та необхідність повторно відправляти втрачені пакети. У геймінгу TCP використовується для операцій, де втрати даних неприпустимі, наприклад, під час авторизації, внутрішньоігрових транзакцій чи збереження прогресу.

Інший базовий протокол - UDP (User Datagram Protocol). На відміну від TCP, він не гарантує доставку всіх пакетів і не контролює їхній порядок.

Натомість він дозволяє передавати дані набагато швидше. І ця швидкість є критично важливою для динамічних ігрових подій: рух персонажів, бойові дії, будь-які зміни, що відбуваються десятки разів на секунду. У такому сценарії, втрата окремого, вже застарілого пакета - значно менш критична, ніж затримка всієї ігрової сесії.

Саме тому більшість сучасних екшенів та шутерів обирають UDP як основний транспорт для обміну даними в реальному часі.

Для наочної ілюстрації принципової різниці між цими двома підходами розглянемо схему передачі даних (рисунок 1.3).

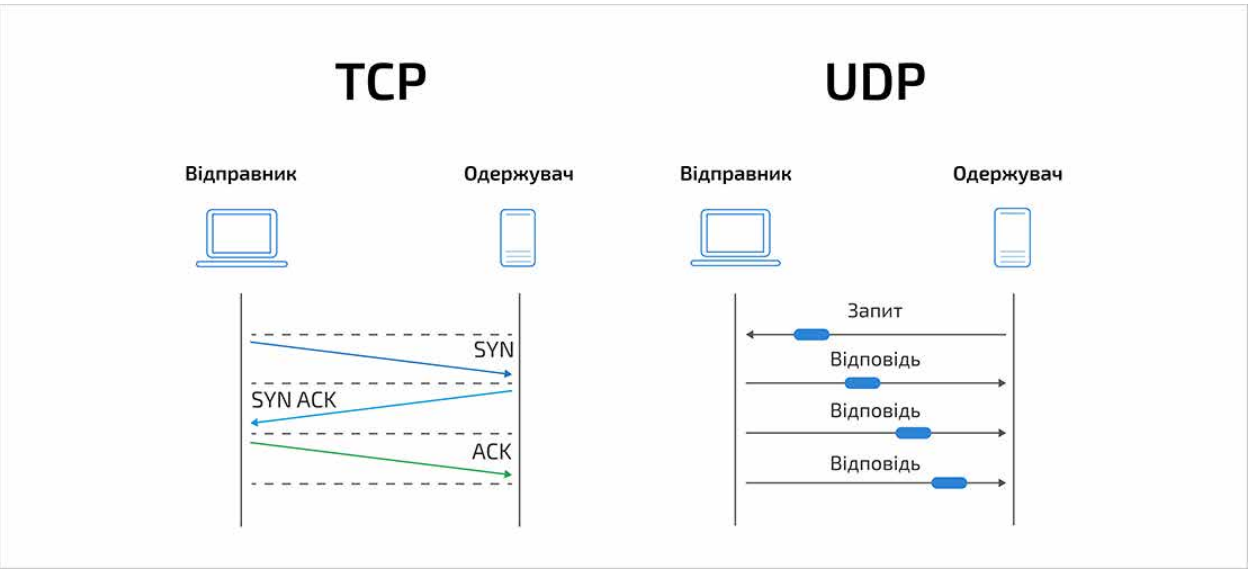

Рисунок 1.3 – Принципова різниця у передачі даних між протоколами TCP та UDP (Адаптовано з [10])

Розвиток вебтехнологій породив нове рішення - WebSocket. Його суть полягає у забезпеченні постійного, двостороннього з’єднання між клієнтом і сервером, що встановлюється на основі TCP. Цей підхід знімає саму необхідність

надсилати численні HTTP-запити. Він робить комунікацію значно ефективнішою. Проте, він неминуче успадковує й певні затримки, властиві його TCP як його базовому протоколу [11].

Тому WebSocket і знайшов своє широке застосування у вебіграх та мобільних клієнтах. Тобто там, де важлива швидка синхронізація, але відсутні ті жорсткі вимоги до мінімізації затримок, які диктують змагальні шутери.

Таким чином, у сучасній ігровій індустрії панує гібридний підхід. TCP використовується для критичних операцій - там, де надійність стоїть на першому місці.Тоді як UDP забезпечує швидкий, динамічний геймплей у реальному часі.

Саме таке поєднання кількох протоколів у межах однієї гри і дає змогу досягти того самого оптимального балансу між надійністю та швидкодією.

1.5. Особливості жанру roguelike у контексті розподілених систем

Жанр roguelike за останнє десятиліття пройшов вражаючий шлях: від нішевого напрямку до одного з найпопулярніших у геймдев-індустрії. Його головними, визначальними ознаками є процедурна генерація рівнів [12], висока варіативність ігрового процесу та наявність постійних викликів для гравця. І, звісно, так званий «перманентний прогрес» - можливість зберігати частину здобутків між окремими ігровими сесіями [13].

Саме ці, здавалося б, суто ігрові особливості, насправді накладають вельми специфічні вимоги на архітектуру ігрових серверів. І ця специфіка стає особливо гострою, коли йдеться про багатокористувацькі або кооперативні проєкти. Найпершою і найважливішою з цих вимог є швидка, «на льоту», генерація унікального контенту. Ця вимога є прямим наслідком того, що рівні створюються процедурно і ніколи не повторюються.

Тому сервер повинен володіти інструментами, які вирішують одразу два завдання: не просто формувати карти за алгоритмами випадковості, але й негайно синхронізувати їх між усіма учасниками сесії. І це, звісно, створює

пряме навантаження на обчислювальні ресурси та висуває гостру потребу у добре продуманій системі кешування.

Другою характерною рисою є велика кількість незалежних ігрових сесій. На відміну від MMO, де тисячі користувачів взаємодіють у єдиному, спільному світі, roguelike-ігри, навпаки, дробляться на безліч окремих «кімнат» або «забігів».

Саме така структура робить архітектуру ідеальною для мікросервісного підходу, за якого кожна окрема сесія може існувати як автономний, ізольований процес, що, тим не менш, залишається керованим з боку центрального оркестратора.

Ще однією важливою вимогою є низька затримка в реальному часі. Незважаючи на всю процедурність і випадковість подій, реакція персонажів на дії гравців повинна залишатися миттєвою. Це означає, що серверна частина повинна підтримувати високопродуктивні мережеві протоколи (переважно UDP) та надійні системи балансування навантаження.

Не менш важливим є збереження прогресу гравця. У багатьох сучасних roguelike (наприклад, Risk of Rain 2 або Vampire Survivors) існує механіка поступового вдосконалення персонажа, навіть після завершення сесії. Це вимагає ефективних рішень для роботи з базами даних, де поєднуються реляційні структури акаунтів і нереляційні - для зберігання результатів забігів.

Таким чином, жанр roguelike можна розглядати як особливий полігон для розвитку розподілених ігрових серверів. Його вимоги до генерації унікального контенту, обслуговування великої кількості паралельних сесій і підтримки мінімальної затримки роблять актуальними використання мікросервісних архітектур, контейнеризації (Docker), оркестрації (Kubernetes) та гібридних баз даних (SQL + NoSQL + кеш).

2. АНАЛІЗ СУЧАСНИХ РІШЕНЬ ТА ТЕХНОЛОГІЙ

2.1. Огляд комерційних платформ для ігрових серверів (AWS GameLift, Photon, Azure PlayFab тощо)

Комерційні платформи для розгортання ігрових серверів останніми роками стали важливим інструментом для студій різного масштабу - від інді-розробників до великих корпорацій. Їхня головна перевага полягає у знятті з розробників необхідності самостійно підтримувати інфраструктуру, що потребує значних ресурсів і спеціалізованих знань. Такі рішення надають готові сервіси для управління сесіями, масштабування, аналітики та безпеки, дозволяючи зосередитися на власне ігровому процесі.

Одним із найпопулярніших рішень є AWS GameLift від Amazon [14]. Ця платформа пропонує автоматичне масштабування серверів залежно від кількості гравців, підтримує різні типи сесій та дозволяє легко інтегрувати анти-чит механізми. Завдяки розподіленим дата-центрам Amazon GameLift забезпечує низьку затримку для користувачів із різних регіонів. Проте використання цієї платформи пов’язане з високою вартістю, особливо при великій кількості активних користувачів, що робить її менш доступною для невеликих команд.

Ще одним поширеним варіантом є Photon Engine, який часто використовується для мобільних ігор і проєктів із невеликими сесіями [15]. Його перевага полягає у простоті впровадження та наявності SDK для Unity та Unreal Engine. Photon добре підходить для реалізації ігор із кооперативним або змагальним мультиплеєром, проте його можливості масштабування обмежені в порівнянні з AWS GameLift.

Azure PlayFab від Microsoft пропонує більш комплексний підхід: крім управління ігровими сесіями та серверною інфраструктурою, ця платформа надає інструменти аналітики, системи монетизації та інтеграцію з Xbox Live. Таким чином, PlayFab є не лише серверним рішенням, а й повноцінною екосистемою для підтримки ігор як сервісу (Games-as-a-Service). Водночас вона

тісно прив’язана до екосистеми Microsoft, що може обмежувати вибір технологій у розробників.

Для порівняння варто згадати ще й Google Cloud Game Servers, які інтегруються з Kubernetes і орієнтовані на студії, що прагнуть більшої гнучкості у розгортанні [16]. Вони дозволяють легко адаптувати кластер під потреби конкретної гри, проте вимагають від розробників вищого рівня технічної компетенції, ніж у випадку Photon чи PlayFab.

Аналізуючи ці рішення, можна дійти висновку, що комерційні платформи надають широкий спектр можливостей, але вибір конкретної залежить від цілей і ресурсів студії. Для інді-проєктів більш доцільними є Photon або PlayFab завдяки простоті інтеграції, тоді як великі студії частіше віддають перевагу AWS GameLift чи Google Cloud через масштабованість та контроль.

2.2. Порівняння відкритих рішень (Colyseus, Mirror, SmartFoxServer, C#/.NET, Node.js WebSocket)

Окрім комерційних платформ, значного поширення набули відкриті серверні рішення, які розробники можуть інтегрувати без додаткових ліцензійних витрат. Вони особливо популярні серед інді-студій та дослідників, оскільки дозволяють повністю контролювати інфраструктуру, адаптувати її під специфічні потреби та мінімізувати витрати.

Ці рішення можна умовно поділити на дві категорії: готові високорівневі фреймворки та базові платформи.

До першої категорії належать такі рішення, як Colyseus, Mirror та SmartFoxServer. Colyseus, створений спеціально для середовища JavaScript/TypeScript, пропонує простоту та швидке розгортання на базі Node.js [17]. Mirror [18] є популярним рішенням, розробленим спеціально для рушія Unity; він забезпечує швидку інтеграцію, але має суттєвий недолік - тісну прив'язку виключно до Unity. SmartFoxServer, у свою чергу, є потужним та перевіреним часом рішенням на базі Java, яке підтримує багатий набір функцій і

орієнтоване на високу масштабованість, проте є значно складнішим у налаштуванні та інтеграції.

До другої категорії належать фундаментальні платформи, такі як Node.js або C#/.NET [19, 20]. Підхід, обраний у даній роботі - розробка на базі Node.js та бібліотеки WebSocket - надає максимальну гнучкість та повний контроль над архітектурою. Альтернативна платформа C#/.NET (з використанням ASP.NET Core) пропонує переваги компільованої мови та багатопотокової архітектури, що робить її ідеальним об'єктом для порівняльного дослідження продуктивності в наступних розділах. Обидва ці платформні підходи вимагають створення всієї інфраструктури з нуля, що збільшує трудовитрати, але дозволяє уникнути будь-яких обмежень, що накладаються сторонніми фреймворками [21].

Для наочного порівняння цих підходів розглянемо їхні ключові характеристики у вигляді таблиці (таблиця 2.1).

Таблиця 2.1

Порівняльний аналіз відкритих серверних рішень

Рішення	Архітектура / Мова	Цільовий рушій	Масштабованість	Складність інтеграції
Colyseus	Фреймворк (JavaScript/TS)	Будь-який	Низька/Середня	Легка
Mirror	Фреймворк (C#)	Тільки Unity	Низька/Середня	Дуже легка
SmartFoxServer	Фреймворк (Java)	Будь-який	Висока	Складна
C#/.NET + WS	Платформа (C#)	Будь-який	Висока (багатопотокова)	Висока (з нуля)
Node.js + WS (обраний підхід)	Платформа (JavaScript/TS)	Будь-який	Залежить від архітектури	Висока (з нуля)

Таким чином, аналіз показує, що хоча готові фреймворки спрощують розробку, створення власного рішення на базовій платформі (Node.js) є виправданим для дослідницьких цілей та проєктів, де потрібен повний контроль над архітектурою та потенціал для побудови унікальної системи масштабування.

2.3. Використання контейнеризації та оркестрації (Docker, Kubernetes)

Контейнеризація та оркестрація стали стандартом у сучасній розробці серверних систем, ігрова інфраструктура не є винятком. Ці технології дозволяють розробникам створювати ізольовані середовища для кожного сервісу та ефективно управляти масштабуванням системи.

Docker [22] використовується для упаковки серверних компонентів у контейнери, які включають все необхідне для їхньої роботи: код, бібліотеки та конфігурації. Завдяки цьому серверні модулі легко розгортати на будь-якому обладнанні або в хмарі, а процес інсталяції стає більш передбачуваним [23]. У випадку ігор це дозволяє швидко запускати нові екземпляри серверів для окремих сесій гравців, що особливо актуально для жанру roguelike, де кожна партія є унікальною.

Однак контейнеризація сама по собі не вирішує проблеми масштабування. Для управління великою кількістю контейнерів використовується Kubernetes [24] - система оркестрації, яка автоматично розподіляє навантаження, відновлює сервіси після збоїв і масштабує їх залежно від кількості користувачів. Наприклад, якщо в певний час доби кількість активних гравців різко зростає, Kubernetes може створити додаткові серверні інстанси без втручання розробників [25].

Поєднання Docker і Kubernetes особливо ефективне для багатокористувацьких ігор, де є потреба в обслуговуванні великої кількості незалежних сесій. Система може гнучко реагувати на зміни навантаження, забезпечуючи стабільну роботу без перевитрат ресурсів. Важливо й те, що ці технології добре інтегруються з хмарними платформами (AWS, Azure, Google

Cloud), які надають готові інструменти для автоматизованого керування контейнерними кластерами [26].

Таким чином, використання контейнеризації та оркестрації стало базовим підходом у сучасній ігровій індустрії. Воно забезпечує незалежність від конкретного обладнання, простоту розгортання, високу відмовостійкість і масштабованість, що робить ці технології ключовими для розробки інфраструктури багатокористувацьких ігор.

2.4. Методи балансування навантаження та відмовостійкість у ігрових системах

Важливою складовою архітектури будь-якого сучасного ігрового сервера є система балансування навантаження та механізми забезпечення відмовостійкості. Від них залежить, чи зможе серверна інфраструктура витримати різкі стрибки кількості користувачів, які є характерними для онлайн-ігор, і забезпечити безперервність роботи навіть у випадку збоїв окремих компонентів.

Балансування навантаження передбачає рівномірний розподіл вхідних запитів від клієнтів між кількома серверними вузлами системи. Замість того, щоб усе навантаження лягало на один-єдиний сервер, створюючи "вузьке місце", спеціальний компонент-балансувальник діє як "диспетчер", спрямовуючи кожного нового гравця на найменш завантажений сервер у кластері [27]. Принцип роботи цього підходу наочно ілюструє рисунок 2.1.

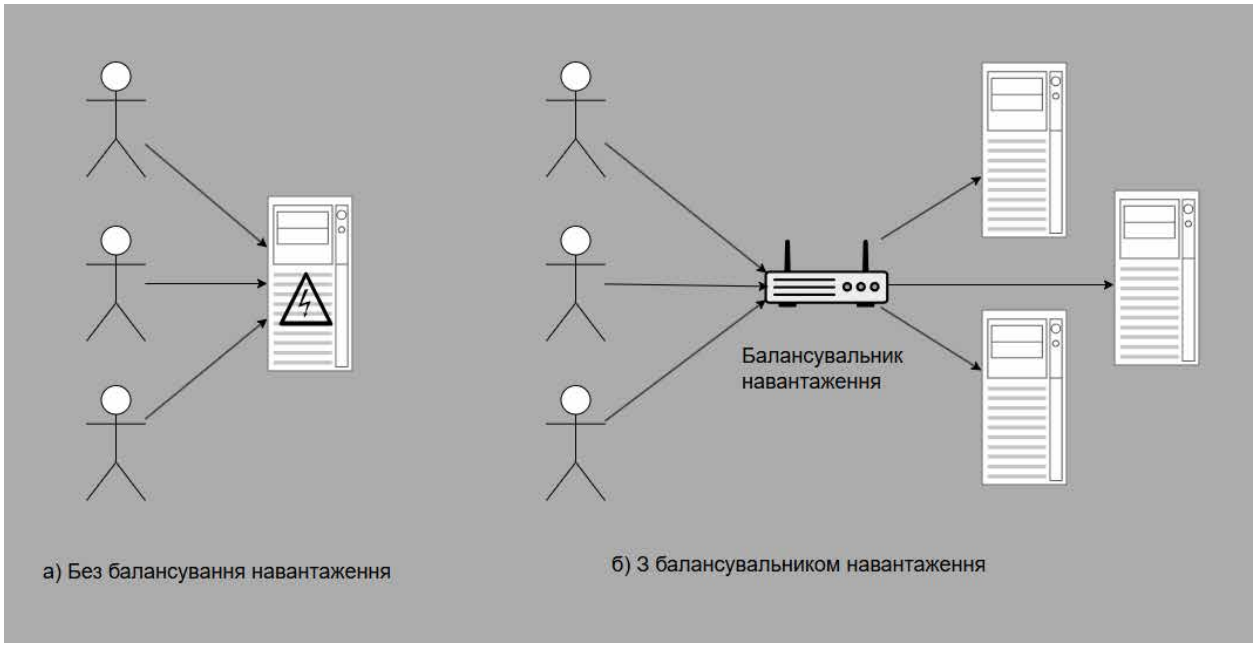

Рисунок 2.1 – Принцип роботи балансувальника навантаження

У багатокористувацьких іграх застосовуються різні методи балансування, такі як рівномірний розподіл (Round-robin), балансування з урахуванням завантаження ресурсів (Resource-based) та географічне балансування для зменшення затримки [28]. У великих проєктах часто поєднуються всі ці методи, щоб забезпечити як високу продуктивність, так і комфорт для гравців у різних регіонах світу.

Відмовостійкість, у свою чергу, передбачає здатність системи продовжувати роботу навіть у разі відмови окремих її компонентів. Це реалізується завдяки таким рішенням, як реплікація даних на кількох вузлах, щоб уникнути їх втрати; автоматичне відновлення контейнерів за допомогою систем оркестрації, які створюють новий екземпляр сервісу замість того, що вийшов з ладу; та failover-механізми, коли резервні сервери автоматично перебирають на себе функції несправних.

Прикладом є інфраструктура ігор жанру MMO, де навіть короткочасний збій може призвести до відключення десятків тисяч гравців. У таких випадках застосовуються складні системи балансування і дублювання даних, що мінімізує ризики й забезпечує стабільність ігрового процесу. Таким чином, балансування навантаження та відмовостійкість є критичними факторами для сучасних ігрових

систем, що гарантують їх стабільність та формують позитивний користувацький досвід.

2.5. Приклади застосування в реальних іграх жанру roguelike

Жанр roguelike має низку особливостей, які безпосередньо впливають на архітектуру серверних систем. Процедурна генерація рівнів, висока варіативність сесій та ізоляція кожної гри формують вимоги до гнучкої та масштабованої інфраструктури. Водночас у суміжних жанрах часто використовуються аналогічні принципи, що дозволяє брати їх як орієнтир для побудови розподілених систем.

Vampire Survivors у своїй початковій версії була орієнтована на офлайн-гру, але розвиток мультиплеєрного режиму показав необхідність інтеграції серверної складової. Основне завдання тут полягає в одночасному запуску сотень незалежних сесій. Це можливо завдяки контейнеризації: кожен забіг існує в окремому серверному інстансі, ізольованому від інших.

У Risk of Rain 2 кооперативна модель гри вимагає синхронізації процедурно створених рівнів і предметів між усіма учасниками. Тут активно використовуються системи балансування трафіку та масштабування серверів, оскільки навантаження може суттєво зростати під час пікових годин. Приклади подібної архітектури можна знайти у Photon або PlayFab, що орієнтовані на багатокористувацькі кооперативні ігри.

Цікавою є й практика суміжних ігор. Наприклад, Deep Rock Galactic не належить до roguelike, проте є кооперативним PvE-проєктом, де рівні генеруються процедурно. Серверна архітектура тут має забезпечувати одночасну роботу груп із 4 гравців у численних інстансах, що подібно до структури roguelike-сесій. Це гарний приклад того, як технології процедурної генерації поєднуються з багатокористувацьким досвідом.

Roboquest поєднує риси roguelite та шутера від першої особи. Тут важливо забезпечити одночасну синхронізацію процедурних рівнів і динамічного бою.

Використання гнучких серверних рішень із низькою затримкою дозволяє реалізувати швидкий мультиплеєр, не втрачаючи характерної для roguelike непередбачуваності.

Окремо варто розглянути приклад Minecraft, який хоч і не є roguelike, але широко застосовує процедурну генерацію світів і підтримує масову багатокористувацьку гру. Серверна інфраструктура Minecraft побудована на принципі розподілу світів і сесій, що дозволяє одночасно утримувати тисячі гравців. Це хороший зразок для вивчення масштабування та керування окремими ігровими просторами.

Ще одним показовим прикладом є Warframe - онлайн-гра з процедурною генерацією місій і високими вимогами до відмовостійкості серверів. Вона поєднує клієнт-серверну модель із розподіленою логікою: частина обчислень виконується локально, а сервери відповідають за синхронізацію прогресу, управління економікою та балансування ігрових сесій. Завдяки такій архітектурі Warframe вдається підтримувати стабільний мультиплеєр для мільйонів користувачів по всьому світу.

Загальний аналіз показує, що серверні системи у roguelike та суміжних жанрах мають виконувати дві ключові функції:

- підтримка унікальних та ізольованих сесій, які можуть масштабуватися залежно від кількості гравців;
- створення мережевих сервісів для синхронізації прогресу, кооперативного або змагального геймплею.

Таким чином, сучасні ігри доводять, що незалежно від жанрової приналежності, розподілені серверні рішення є необхідною умовою для масштабування ігрового процесу та інтеграції соціальних елементів у спільноту.

3. РОЗРОБКА ТА МОДЕЛЮВАННЯ ВДОСКОНАЛЕНОЇ АРХІТЕКТУРИ РОЗПОДІЛЕНОГО ІГРОВОГО СЕРВЕРА

3.1. Постановка задачі та вимоги до системи

На основі проведеного в попередніх розділах аналізу теоретичних основ розподілених систем, сучасних архітектурних підходів та специфіки жанру roguelike, можна перейти до практичної частини роботи. Метою даного етапу є формулювання конкретних завдань та визначення ключових вимог до експериментальної серверної системи. Цей процес є критично важливим, оскільки він закладає фундамент для подальшого проєктування та розробки, визначаючи як функціональні можливості, так і нефункціональні атрибути якості, такі як продуктивність, надійність та масштабованість. Завдання полягає у створенні прототипу розподіленої серверної інфраструктури, здатної ефективно обслуговувати багатокористувацьку гру в жанрі roguelike, забезпечуючи при цьому стабільний та плавний ігровий процес для кінцевих користувачів.

Функціональні вимоги до системи визначають, які саме операції та завдання вона повинна виконувати. Першою і ключовою вимогою є управління ігровими сесіями. Система повинна бути здатною створювати, підтримувати та коректно завершувати велику кількість одночасних ігрових сесій для різних груп гравців. Це означає, що для кожної партії має бути виділений окремий, ізольований ігровий простір, який не впливає на інші активні ігри. Сервер повинен надійно обробляти підключення нових гравців, їх вихід із сесії та забезпечувати стабільність ігрового циклу протягом усього "забігу".

Другою важливою вимогою, що випливає зі специфіки жанру, є процедурна генерація контенту. Серверна частина повинна містити алгоритмічний апарат для створення унікальних ігрових рівнів для кожної нової сесії. Цей процес включає не лише генерацію топології карти, а й осмислене розміщення на ній ключових елементів: скарбів, ворогів, а також точок старту та

фінішу. Алгоритми генерації мають бути достатньо швидкими, щоб гравець не відчував затримки на початку гри, і водночас достатньо варіативними, щоб забезпечити високий рівень реіграбельності.

Наступною вимогою є синхронізація стану в реальному часі. Оскільки гра є багатокористувацькою, сервер повинен ефективно збирати дані про дії всіх гравців (зміна позиції, взаємодія з об'єктами) та розсилати оновлений стан гри всім учасникам сесії з мінімальною затримкою. Це критично важливо для забезпечення плавного та чесного ігрового процесу, де всі гравці бачать узгоджену картину ігрового світу. Для цього необхідно обрати відповідні мережеві протоколи та розробити ефективний формат обміну даними.

Крім того, система повинна забезпечувати обробку ключових ігрових подій. До таких подій належать збір гравцем скарбів, що має призводити до оновлення його рахунку; зіткнення з ворогами, яке повинно викликати відповідну реакцію (наприклад, повернення на старт); а також досягнення фінішу рівня, що ініціює завершення поточної сесії та генерацію нової карти. Логіка обробки цих подій має бути реалізована виключно на сервері, щоб унеможливити шахрайство з боку клієнта.

Останньою функціональною вимогою є персистентність даних. Прогрес гравців, зокрема їхній рахунок, не повинен втрачатися після завершення ігрової сесії або виходу з гри. Для цього система повинна взаємодіяти з базою даних, зберігаючи та оновлюючи ключову інформацію про акаунти користувачів. Це дозволяє реалізувати елементи мета-прогресії, характерні для сучасних ігор жанру roguelike.

Нефункціональні вимоги описують атрибути якості системи та обмеження, в рамках яких вона має працювати. Найважливішою з них є масштабованість. Архітектура має бути спроєктована таким чином, щоб її можна було легко розширювати для обслуговування зростаючої кількості гравців. Перевага надається горизонтальному масштабуванню, тобто можливості додавати нові серверні вузли, а не нарощувати потужність існуючого. Це дозволяє гнучко реагувати на пікові навантаження та оптимізувати витрати на інфраструктуру.

Не менш важливою є відмовостійкість. Система повинна продовжувати стабільно працювати навіть у випадку збою окремих її компонентів. Наприклад, помилка, що призвела до "падіння" однієї ігрової сесії, не повинна впливати на роботу інших сесій або всієї системи в цілому. Це досягається шляхом декомпозиції архітектури та механізмів автоматичного відновлення.

Нарешті, ключовою вимогою є ізоляція сесій. Кожна ігрова сесія повинна функціонувати в ізольованому програмному середовищі, не маючи доступу до даних чи ресурсів інших сесій. Це необхідно як з точки зору безпеки, так і для забезпечення стабільної продуктивності, оскільки проблеми в одній грі не зможуть сповільнити чи пошкодити інші.

3.2. Вибір архітектурного підходу та технологічного стеку

Для ефективного досягнення цілей, сформульованих у попередньому підрозділі, необхідно було зробити обґрунтований вибір як загальної архітектурної моделі, так і конкретного набору інструментів для її реалізації. Базуючись на аналізі, проведеному в теоретичній частині роботи, було вирішено зупинитися на класичній клієнт-серверній архітектурі з авторитетним сервером. Ця модель є де-факто стандартом для більшості сучасних онлайн-ігор, оскільки вона надає ключову перевагу: централізований контроль над ігровою логікою. У такій парадигмі клієнтський застосунок відповідає переважно за візуалізацію та збір даних від користувача, тоді як сервер є "єдиним джерелом правди", що приймає всі рішення щодо стану ігрового світу. Такий підхід значно ускладнює спроби шахрайства з боку гравців (чітерства) та гарантує, що всі учасники сесії бачать консистентну та узгоджену картину подій.

Водночас, з огляду на вимогу до ізоляції великої кількості незалежних сесій, що є характерною рисою жанру roguelike, чиста клієнт-серверна модель була доповнена елементами мікросервісного підходу. Замість того, щоб усі ігрові сесії виконувалися в рамках єдиного монолітного процесу, кожна сесія була спроєктована як окремий, логічно ізольований компонент. Такий гібридний

підхід дозволяє поєднати надійність і керованість централізованої моделі з гнучкістю та відмовостійкістю, властивими мікросервісам. Збій або надмірне навантаження в одній ігровій сесії не вплине на стабільність інших, що є критично важливим для забезпечення якості користувацького досвіду.

Вибір технологічного стеку було зроблено з урахуванням вимог до продуктивності, швидкості розробки та потенціалу для подальшого масштабування. Для серверної частини було обрано платформу Node.js [29] у зв'язці з бібліотекою WebSocket. Це рішення є оптимальним для обробки великої кількості одночасних з'єднань у реальному часі. Асинхронна, керована подіями архітектура Node.js дозволяє ефективно управляти мережевими запитами без блокування основного потоку виконання, що забезпечує високу пропускну здатність та низьку затримку, що є критичним для динамічних ігор.

Для клієнтської частини було обрано ігровий рушій Unity [30], оскільки він є одним із лідерів індустрії, має потужні інструменти для роботи з 3D-графікою та фізикою, а також широку підтримку спільноти та наявність готових бібліотек для інтеграції мережевих протоколів, зокрема WebSocket.

З метою забезпечення ізоляції сесій та уніфікації середовища розгортання було прийнято рішення про використання технології контейнеризації Docker [31]. Упаковка серверного застосунку в Docker-контейнер дозволяє уникнути проблем сумісності, гарантуючи, що код буде працювати однаково як на локальній машині розробника, так і на будь-якому серверному обладнанні. Це також є фундаментальним кроком, що закладає основу для майбутнього впровадження систем оркестрації для автоматичного масштабування.

3.3. Детальна архітектура спроєктованої системи

Після визначення загального підходу та технологічного стеку, наступним етапом є детальне проєктування архітектури системи. Цей підрозділ представляє конкретну структуру та взаємозв'язки між компонентами, що були розроблені для досягнення поставлених функціональних та нефункціональних вимог.

Спроєктована архітектура є практичною реалізацією гібридної моделі, де надійність класичної клієнт-серверної парадигми поєднується з гнучкістю та ізоляцією, властивими мікросервісному підходу. Метою було створення не монолітного застосунку, а модульної системи, де кожен компонент має чітко визначену зону відповідальності, що спрощує подальшу підтримку, тестування та розширення.

Для наочної візуалізації спроєктованої архітектури та взаємозв'язків між її компонентами було розроблено відповідну схему (рисунок 3.1).

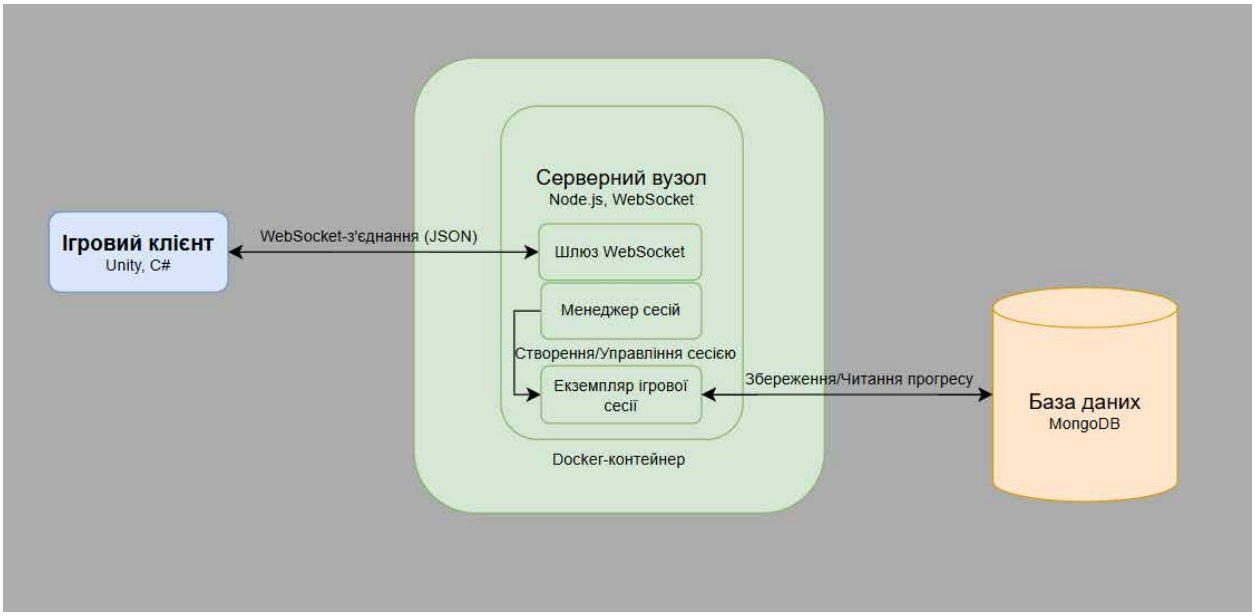

Рисунок 3.1 – Загальна архітектура розподіленої системи

Загальну структуру системи, як показано на схемі, можна представити у вигляді трьох основних логічних рівнів: клієнтський рівень, серверний рівень та рівень зберігання даних.

Основні компоненти системи:

- Ігровий клієнт (Unity): Цей компонент є точкою взаємодії користувача з ігровим світом. Його основна відповідальність полягає у візуалізації графіки, відтворенні звуків та обробці вводу від гравця (наприклад, натискання клавіш для руху). У рамках авторитетної серверної архітектури клієнт не приймає жодних рішень щодо ігрової логіки. Він лише збирає дії гравця, надсилає їх на сервер для обробки, а потім отримує та відображає

оновлений стан ігрового світу. Такий підхід робить клієнт "тонким", мінімізуючи обчислювальне навантаження на нього та ускладнюючи спроби маніпуляції ігровим процесом.

- Серверний вузол (Node.js): Це ядро всієї системи, що виконує всю обчислювальну логіку. Хоча він запускається як єдиний застосунок, його внутрішня структура є модульною, що дозволяє чітко розмежувати обов'язки. Він складається з кількох взаємопов'язаних підсистем:
    - Шлюз WebSocket: Виступає як єдина точка входу для всіх клієнтських з'єднань. Він відповідає за встановлення, підтримку та закриття WebSocket-з'єднань, а також за прийом та відправку сирих даних по мережі.
    - Менеджер сесій: Цей компонент діє як внутрішній оркестратор. Коли новий гравець підключається до сервера, менеджер сесій створює для нього новий екземпляр ігрової сесії (або додає до існуючої, якщо реалізовано кооперативний режим) та асоціює з'єднання гравця з цією сесією.
    - Екземпляр ігрової сесії: Це логічно ізольований об'єкт, який можна розглядати як "мікросервіс" для однієї конкретної гри. Він містить повний стан однієї партії: процедурно згенеровану карту, поточні позиції, рахунки та інші атрибути всіх гравців у цій сесії. Саме цей компонент обробляє всі ігрові події, такі як TAKE_TREASURE або HIT_ENEMY, оновлює стан гри відповідно до своєї логіки та розсилає оновлення всім учасникам сесії.
- Контейнер Docker: Весь серверний вузол Node.js разом з усіма його системними залежностями пакується в єдиний Docker-образ. Цей підхід забезпечує максимальну портативність та відтворюваність середовища. Завдяки контейнеризації серверний застосунок можна розгорнути на будь-якій машині, де встановлено Docker, без необхідності в ручному налаштуванні. Це є практичною реалізацією вимоги до ізоляції, оскільки кожен запущений контейнер працює у власному ізольованому просторі.

- База даних (MongoDB) [32]: Винесена в окремий, незалежний сервіс, що відповідає за довготривале зберігання даних. На відміну від ігрових сесій, які зберігають тимчасовий стан гри в оперативній пам'яті, база даних забезпечує персистентність. Вона зберігає інформацію про профілі гравців, їхній глобальний прогрес, рекорди та, у нашому випадку, загальний рахунок скарбів. Взаємодія ігрових сесій з базою даних відбувається асинхронно, переважно в кінці гри, щоб операції запису на диск не впливали на продуктивність ігрового процесу в реальному часі.

Життєвий цикл ігрової сесії в рамках спроєктованої архітектури виглядає наступним чином. Спочатку клієнт ініціює WebSocket-з'єднання з сервером. Шлюз приймає з'єднання і передає його менеджеру сесій, який створює новий екземпляр ігрової сесії. Цей екземпляр генерує унікальну карту та надсилає її клієнту разом з початковими даними. Протягом гри клієнт надсилає на сервер короткі повідомлення про дії гравця. Ігрова сесія обробляє ці повідомлення, оновлює свій внутрішній стан (наприклад, змінює координати гравця) і розсилає актуалізовані дані всім учасникам. Після завершення гри сесія надсилає фінальний результат у базу даних MongoDB для збереження, після чого екземпляр сесії може бути знищений, звільняючи ресурси сервера.

3.4. Протоколи взаємодії та формати даних

Ефективна та надійна комунікація між ігровим клієнтом та сервером є однією з ключових умов для забезпечення стабільного ігрового процесу в реальному часі. Для цього необхідно розробити чіткий та добре структурований протокол взаємодії, який визначає як формат повідомлень, так і логіку їх обміну. Враховуючи, що взаємодія відбувається через постійне двостороннє з'єднання WebSocket, протокол має бути легковажним, щоб мінімізувати мережеві затримки, та водночас достатньо гнучким, щоб передавати різноманітні ігрові дані - від координат гравців до складних структур, що описують карту.

Як основний формат серіалізації даних було обрано JSON (JavaScript Object Notation). Цей вибір обґрунтований кількома вагомими перевагами. По-перше, JSON є текстовим форматом, що робить його легкочитним для людини, що значно спрощує процеси розробки та відлагодження. По-друге, він нативно підтримується середовищем Node.js, що дозволяє серверу обробляти вхідні повідомлення з мінімальними витратами на парсинг. По-третє, для клієнтської частини на Unity існує безліч високоефективних бібліотек (зокрема, Newtonsoft.Json), які дозволяють швидко та надійно серіалізувати та десеріалізувати C#-об'єкти в JSON-рядки. Хоча бінарні формати, такі як Protocol Buffers, можуть забезпечити вищу щільність даних, для прототипу гри жанру roguelike переваги JSON у простоті та швидкості розробки є більш значущими.

Кожне повідомлення, що передається між клієнтом та сервером, є JSON-об'єктом, який обов'язково містить текстове поле type. Це поле діє як ідентифікатор команди або події, дозволяючи приймаючій стороні зрозуміти, як саме інтерпретувати та обробляти решту даних у повідомленні. Такий підхід створює просту, але ефективну систему маршрутизації повідомлень. Для наочної демонстрації логіки взаємодії розглянемо діаграму послідовності для одного з ключових ігрових сценаріїв - підбору скарбу гравцем (рисунок 3.2).

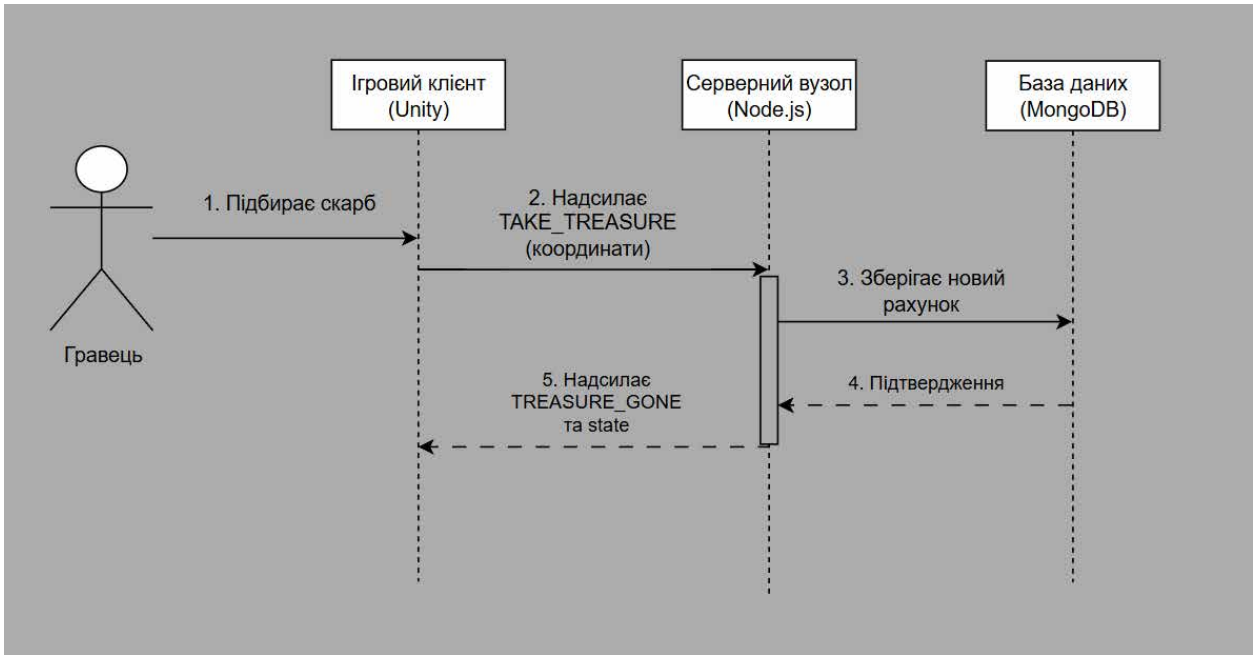

Рисунок 3.2 – Діаграма послідовності для сценарію "Підбір скарбу"

Повідомлення від клієнта до сервера

Клієнтський застосунок ініціює взаємодію, надсилаючи на сервер повідомлення про дії гравця. Ці повідомлення є короткими та містять лише необхідну для обробки інформацію, щоб мінімізувати вихідний трафік.

- Оновлення позиції гравця: Повідомлення POS:x,y,z є найчастішим. Воно надсилається, коли гравець рухається. Для максимальної оптимізації дані про координати передаються у вигляді простого рядка, що зменшує накладні витрати на JSON-структуру.
- Взаємодія з об'єктами: Коли гравець намагається підібрати скарб, клієнт надсилає повідомлення TAKE_TREASURE з JSON-об'єктом, що містить координати цільового скарбу. Аналогічно, при зіткненні з ворогом надсилається повідомлення HIT_ENEMY, а при досягненні кінця рівня - FINISH. Всі ці повідомлення є запитами на зміну стану гри, рішення щодо яких приймає виключно сервер.

Повідомлення від сервера до клієнта

Сервер, у свою чергу, надсилає клієнту повідомлення, які інформують його про стан ігрового світу та результати його дій. Ці повідомлення можуть бути як реакцією на запит клієнта, так і проактивними сповіщеннями для всіх учасників сесії.

- Привітальні дані: Одразу після успішного підключення сервер надсилає клієнту унікальне повідомлення welcome. Воно містить згенерований для гравця clientId та його стартову позицію на карті, що дозволяє клієнту правильно ініціалізувати ігровий процес.
- Дані ігрового світу: Повідомлення map є найбільшим за обсягом і містить повну структуру згенерованого рівня: масив блоків, координати скарбів та ворогів. Воно надсилається на початку кожної нової гри.
- Оновлення стану: Повідомлення state є ключовим для синхронізації. Сервер періодично розсилає його всім гравцям сесії, і воно містить актуальні дані про всіх учасників: їхні позиції, рахунки та інші важливі атрибути.

- Сповіщення про події: Для оптимізації трафіку деякі зміни передаються не через повний стан, а через окремі події. Як показано на діаграмі послідовності (рисунок 3.2), коли один з гравців підбирає скарб, сервер надсилає всім повідомлення TREASURE_GONE з координатами зниклого об'єкта. Це дозволяє клієнтам миттєво оновити сцену, не чекаючи наступного повного оновлення стану.

Такий структурований протокол обміну повідомленнями забезпечує надійну та ефективну комунікацію, що є фундаментом для стабільної роботи багатокористувацької гри в реальному часі.

3.5. Перспективи масштабування та відмовостійкості

Розроблений у рамках даної роботи прототип, який функціонує як єдиний Docker-контейнер, є не кінцевим продуктом, а радше фундаментальною архітектурною одиницею. Цінність обраного підходу, що базується на контейнеризації, полягає не стільки у спрощенні локального розгортання, скільки у величезному потенціалі для подальшого розширення та створення повноцінної промислової системи. Спроєктована архітектура з самого початку створювалася з урахуванням майбутніх потреб у масштабованості та надійності, що є критичними для будь-якого комерційно успішного онлайн-продукту. Цей підрозділ розглядає логічні наступні кроки, які дозволять трансформувати прототип у високоефективну та відмовостійку розподілену систему.

Першим і найважливішим напрямком для розвитку є забезпечення горизонтального масштабування. Цей підхід передбачає збільшення загальної продуктивності системи не за рахунок нарощування потужності одного сервера (вертикальне масштабування), а шляхом додавання нових, ідентичних серверних вузлів. Завдяки контейнеризації Docker, цей процес є максимально стандартизованим: для збільшення потужності достатньо запустити декілька екземплярів одного й того ж контейнера з серверним застосунком. Однак для ефективного управління цими екземплярами необхідно впровадити ключовий

компонент - балансувальник навантаження (наприклад, Nginx або HAProxy). Він діятиме як єдина точка входу для всіх клієнтських з'єднань, виконуючи роль "диспетчера", який інтелектуально розподіляє нових гравців між доступними серверними інстансами. Це дозволить не тільки обслуговувати значно більшу кількість одночасних сесій, але й уникнути ситуації, коли один вузол перевантажений, тоді як інші простоюють.

Проте, просте додавання нових вузлів не вирішує проблему надійності. Для забезпечення високого рівня відмовостійкості та автоматизації управління життєвим циклом контейнерів наступним логічним етапом є впровадження системи оркестрації. Найбільш поширеним галузевим стандартом для цих завдань є Kubernetes. Впровадження цієї системи дозволить перейти від ручного управління контейнерами до повністю автоматизованого процесу.

Основні переваги, які надасть оркестрація:

- Автоматичне відновлення (Self-healing): Kubernetes постійно відстежує стан запущених контейнерів. У випадку, якщо якийсь із серверних екземплярів вийде з ладу через непередбачену помилку в коді або апаратний збій, система автоматично виявить це і негайно запустить новий, здоровий контейнер на заміну. Для кінцевих користувачів, підключених до інших сесій, цей процес залишиться абсолютно непомітним, що забезпечить безперервність ігрового процесу.
- Автоматичне масштабування (Autoscaling): Систему можна налаштувати таким чином, щоб вона самостійно реагувала на зміни навантаження [33]. Наприклад, Kubernetes може моніторити використання центрального процесора на активних вузлах. Якщо навантаження перевищує заданий поріг (наприклад, у вечірні пікові години), система автоматично додасть нові контейнери для обробки зростаючого потоку гравців. Коли ж навантаження спаде (наприклад, вночі), зайві контейнери будуть автоматично зупинені для економії ресурсів.

Таким чином, хоча поточна практична реалізація обмежується прототипом в одному Docker-контейнері, сам вибір архітектурного підходу є стратегічним.

Він закладає правильний і сучасний фундамент для створення надійної, високоефективної та економічно вигідної розподіленої системи, готової до викликів промислової експлуатації.

3.6. Безпекові аспекти та захист від мережевих атак

При проєктуванні будь-якої системи, що передбачає мережеву взаємодію, питання безпеки є одним із найвищих пріоритетів. Ігрові сервери не є винятком, оскільки вони є потенційною ціллю для різноманітних атак, що можуть варіюватися від спроб шахрайства з боку недобросовісних гравців до цілеспрямованих атак, спрямованих на порушення роботи всієї інфраструктури. Недостатня увага до аспектів безпеки може призвести не лише до погіршення ігрового досвіду для чесних гравців, але й до фінансових та репутаційних втрат. Тому в рамках проєктування архітектури було закладено базові принципи та механізми, спрямовані на мінімізацію основних ризиків.

Основною загрозою для будь-якої онлайн-гри є шахрайство (чітерство) з боку гравців. Це може включати маніпуляції з клієнтським кодом для отримання нечесних переваг, таких як збільшення швидкості руху, проходження крізь стіни або миттєвий збір усіх скарбів на карті. Фундаментальним архітектурним рішенням для протидії таким загрозам є використання авторитетного сервера, що було закладено в основу спроєктованої системи. У цій моделі клієнтський застосунок не має жодних повноважень приймати рішення щодо стану гри; він лише надсилає на сервер запити про наміри гравця. Вся ігрова логіка, включаючи перевірку можливості руху, обробку зіткнень та нарахування очок, виконується виключно на сервері. Це робить більшість спроб клієнтських маніпуляцій неефективними, оскільки сервер просто відхилить будь-які дії, що порушують правила гри.

Для забезпечення надійності цього підходу сервер повинен виконувати валідацію всіх вхідних даних від клієнта. Не можна сліпо довіряти інформації, що надходить від гравця. Наприклад, при отФриманні повідомлення про зміну

позиції POS:x,y,z сервер повинен перевіряти, чи є новий стан фізично досяжним з попередньої позиції за минулий проміжок часу, враховуючи встановлену в грі швидкість персонажа. Якщо гравець повідомляє про переміщення на надто велику відстань, такий запит має ігноруватися або призводити до застосування санкцій.

Іншою поширеною загрозою є атаки на відмову в обслуговуванні (Denial-of-Service, DoS). У найпростішому вигляді зловмисник або несправний клієнт може намагатися "заспамити" сервер величезною кількістю повідомлень, щоб вичерпати його обчислювальні ресурси та зробити недоступним для інших гравців. Для протидії цьому в архітектуру закладаються такі механізми:

- Обмеження частоти запитів (Rate Limiting): Сервер повинен відстежувати кількість повідомлень, що надходять від кожного клієнта за певний проміжок часу. Якщо гравець перевищує встановлений ліміт (наприклад, надсилає понад 100 запитів на секунду), його з'єднання може бути тимчасово заблоковане або повністю розірване.
- Захист на рівні інфраструктури: Хоча прототип не включає складних систем захисту, спроєктована архітектура, що передбачає розгортання в хмарному середовищі, дозволяє використовувати потужні інструменти для протидії розподіленим атакам на відмову в обслуговуванні (DDoS). Провайдери хмарних послуг, такі як AWS, Google Cloud або спеціалізовані сервіси на кшталт Cloudflare, надають готові рішення для фільтрації шкідливого трафіку ще до того, як він досягне ігрового сервера.

Таким чином, хоча реалізований прототип має базовий рівень захисту, його архітектура спроєктована з урахуванням сучасних вимог до безпеки та закладає правильний фундамент для побудови надійної та захищеної системи, здатної протистояти основним загрозам у багатокористувацьких онлайн-іграх.

4. ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ

4.1. Вибір інструментів та середовища розробки

Перехід від теоретичного проєктування до практичної реалізації вимагає ретельного вибору конкретних програмних інструментів та налаштування ефективного середовища розробки. Вибір технологій на цьому етапі ґрунтувався на критеріях продуктивності, швидкості розробки, доступності документації та відповідності до архітектурних рішень, прийнятих у попередньому розділі. Метою було створення функціонального прототипу, що дозволяє провести експериментальні дослідження та підтвердити життєздатність спроєктованої системи.

Для серверної частини було обрано платформу Node.js. Цей вибір є стратегічним, оскільки її асинхронна, однопотокова модель, керована подіями, ідеально підходить для обробки великої кількості одночасних WebSocket-з'єднань з мінімальними накладними витратами. Для реалізації мережевої взаємодії була використана популярна бібліотека ws, яка надає низькорівневу, але високоефективну реалізацію протоколу WebSocket [34]. Хоча основна комунікація відбувається через WebSockets, для початкової ініціалізації HTTP-сервера, до якого підключається WebSocket-сервер, було використано мінімалістичний фреймворк Express.js. Це є стандартною та надійною практикою в екосистемі Node.js.

Для клієнтської частини було обрано ігровий рушій Unity (версія 2022.3 LTS). Це рішення обґрунтоване його кросплатформеністю, потужними інструментами для роботи з 3D-графікою та наявністю розвиненої екосистеми ассетів і бібліотек. Основною мовою програмування клієнтської логіки стала C#. Для забезпечення зв'язку з сервером була інтегрована бібліотека NativeWebSocket, яка забезпечує стабільну роботу WebSocket-з'єднання на різних цільових платформах. Для обробки JSON-повідомлень, що надходять від сервера, було використано бібліотеку Newtonsoft.Json (Json.NET) -

високопродуктивне рішення для серіалізації та десеріалізації даних, що дозволяє легко перетворювати JSON-рядки на C#-об'єкти і навпаки.

Для зберігання персистентних даних, зокрема рахунків гравців, було обрано нереляційну систему управління базами даних MongoDB. Її документо-орієнтована модель, що працює з BSON (бінарним представленням JSON), ідеально інтегрується з серверною логікою на Node.js, спрощуючи операції збереження та читання ігрових даних без необхідності у складних схемах чи міграціях.

Нарешті, ключовим елементом середовища розгортання стала технологія контейнеризації Docker. Серверний застосунок Node.js разом з усіма залежностями було упаковано в Docker-контейнер, що дозволило створити повністю ізольоване, портативне та відтворюване середовище. Такий підхід не тільки спростив процес розробки та тестування, але й реалізував перший і найважливіший крок на шляху до створення масштабованої системи, готової до розгортання в кластерному середовищі.

Весь процес розробки вівся з використанням системи контролю версій Git, що забезпечувало відстеження змін та можливість спільної роботи. В якості інтегрованих середовищ розробки використовувалися Visual Studio Code для серверної частини та для клієнта на Unity.

4.2. Опис реалізації ключових модулів системи

Практична реалізація спроєктованої архітектури вимагала розробки кількох взаємопов'язаних програмних модулів, що разом утворюють ядро серверного застосунку та клієнтської частини. Цей підрозділ присвячений детальному огляду ключових компонентів системи.

Результатом роботи клієнтської частини, розробленої на рушії Unity, є повноцінний ігровий процес, що дозволяє гравцю взаємодіяти з процедурно згенерованим світом. Загальний вигляд ігрового клієнта під час активної сесії представлено на рисунку 4.1.

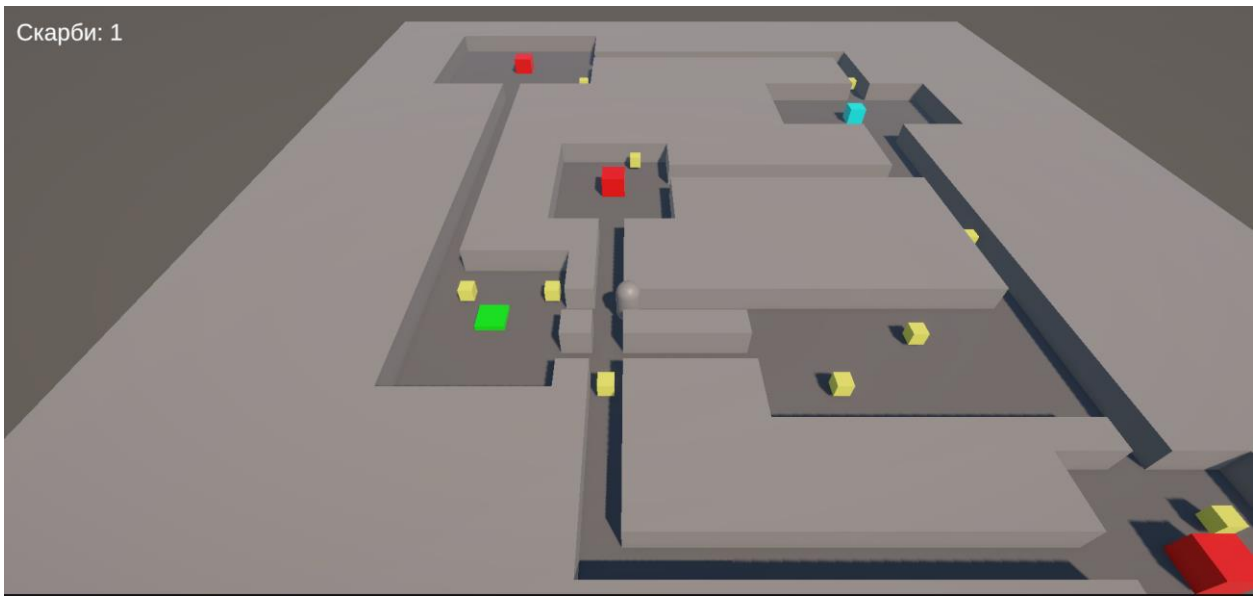

Рисунок 4.1 – Вигляд ігрового процесу на клієнті Unity

На скріншоті видно основні елементи: процедурно згенерований лабіринт, персонаж гравця, об'єкти ворогів (червоні куби) та скарбів (жовті куби), а також елемент інтерфейсу, що відображає поточний рахунок скарбів.

Робота серверної частини супроводжується детальним логуванням ключових подій у консолі, що спрощує процес відлагодження та моніторингу. Приклад логів роботи сервера під час підключення гравця, генерації карти та обробки ігрових подій наведено на рисунку 4.2.

```
game-ws-server | --- Map Generated: 4 enemies found. ---
game-ws-server | 🚀 HTTP + WS server running on port 8080
game-ws-server | ✅ Client connected: 61537b1f-43ef-4764-9461-6a1f75eccb95
game-ws-server | 💰 Treasure at (21, 16) taken by 61537b1f-43ef-4764-9461-6a1f75eccb95. New score: 1
game-ws-server | 💰 Treasure at (21, 18) taken by 61537b1f-43ef-4764-9461-6a1f75eccb95. New score: 2
game-ws-server | 💰 Treasure at (17, 15) taken by 61537b1f-43ef-4764-9461-6a1f75eccb95. New score: 3
game-ws-server | 💰 Treasure at (17, 12) taken by 61537b1f-43ef-4764-9461-6a1f75eccb95. New score: 4
game-ws-server | - Player 61537b1f-43ef-4764-9461-6a1f75eccb95 hit an enemy. Resetting...
game-ws-server | 💰 Treasure at (14, 20) taken by 61537b1f-43ef-4764-9461-6a1f75eccb95. New score: 1
game-ws-server | 💰 Treasure at (13, 20) taken by 61537b1f-43ef-4764-9461-6a1f75eccb95. New score: 2
game-ws-server | 🏁 FINISH received from 61537b1f-43ef-4764-9461-6a1f75eccb95. Regenerating map...
game-ws-server | --- Map Generated: 3 enemies found. ---
game-ws-server | 💰 Treasure at (6, 11) taken by 61537b1f-43ef-4764-9461-6a1f75eccb95. New score: 3
game-ws-server | 🏁 FINISH received from 61537b1f-43ef-4764-9461-6a1f75eccb95. Regenerating map...
game-ws-server | --- Map Generated: 4 enemies found. ---
```

Рисунок 4.2 – Логи роботи ігрового сервера в консолі Node.js

На рисунку видно повідомлення про успішне підключення нового клієнта (Client connected), результат роботи генератора карт (Map Generated: X enemies found), а також обробку події підбору скарбу із зазначенням ID гравця та його

нового рахунку. Це демонструє коректну роботу серверної логіки в реальному часі.

Реалізація сервера ігрових сесій

Сервер ігрових сесій є центральним компонентом системи. Його було реалізовано як застосунок Node.js, що використовує бібліотеку ws для створення WebSocket-сервера. Основна логіка зосереджена в обробнику події connection, який спрацьовує при підключенні нового клієнта. У цей момент для клієнта генерується унікальний ID, створюється початковий стан (координати, рахунок) та ініціюється нова ігрова сесія з генерацією унікальної карти. Подальша логіка реалізована в обробнику події message, який аналізує поле type кожного вхідного JSON-повідомлення та виконує відповідні дії, оновлюючи стан гри та розсилаючи його всім учасникам сесії.

Реалізація модуля процедурної генерації карт

Модуль генерації карт реалізовано у вигляді окремої функції generateSewerMap. Алгоритм є варіацією класичного методу "кімнати та коридори" (Rooms and Corridors). Процес складається з кількох етапів: створення кімнат випадкового розміру та положення; з'єднання їхніх центрів коридорами для забезпечення зв'язності рівня; та розміщення ключових об'єктів (старту, фінішу, ворогів та скарбів) на згенерованій карті. В результаті роботи цього модуля формується об'єкт, що повністю описує стан ігрового світу і готовий для відправки клієнту у форматі JSON.

4.3. Розробка методики та проведення експериментального тестування продуктивності

Після успішної реалізації ключових модулів системи наступним логічним етапом є проведення експериментальних досліджень для об'єктивної оцінки продуктивності та стабільності. Метою цього експерименту є порівняльне кількісне вимірювання ключових показників роботи двох серверних прототипів - однопотокової архітектури на Node.js (Система А) та багатопотокової на

C#/.NET (Система Б) - під ідентичним симульованим навантаженням. Отримані дані дозволять не лише підтвердити життєздатність обраних архітектур, але й виявити та порівняти їхні потенційні "вузькі місця", які можуть стати обмежувальними факторами при подальшому масштабуванні.

Для забезпечення відтворюваності та валідності результатів було розроблено чітку методику тестування, що включає визначення єдиних метрик, опис тестового стенда та ідентичних сценаріїв навантаження для обох систем.

Для всебічної оцінки продуктивності було обрано три ключові метрики, які найкраще характеризують як якість ігрового досвіду для кінцевого користувача, так і ефективність використання серверних ресурсів .

Основні метрики для вимірювання:

- Середній час відгуку сервера (Latency): Цей показник вимірює затримку між відправкою клієнтом повідомлення про дію (зміну позиції) та отриманням відповідного оновлення стану від сервера. Це найважливіший індикатор продуктивності в реальному часі, оскільки висока затримка безпосередньо впливає на плавність ігрового процесу.
- Навантаження на центральний процесор (CPU Load): Ця метрика показує, який відсоток обчислювальної потужності процесора споживає серверний застосунок при різній кількості підключених гравців. Моніторинг цього показника дозволяє визначити, наскільки інтенсивно серверна логіка використовує ресурси та де знаходиться поріг, після якого продуктивність починає деградувати.
- Використання оперативної пам'яті (Memory Usage): Ця метрика відстежує обсяг оперативної пам'яті, що виділяється для серверного процесу. Вона дозволяє оцінити ефективність управління пам'яттю та розрахувати витрати ресурсів на підтримку ігрових сесій.

Для проведення експерименту було розгорнуто тестове середовище, що складалося з однієї фізичної машини, на якій одночасно запускався як тестований сервер (Система А або Система Б), так і клієнти-симулятори. Такий підхід дозволив мінімізувати вплив зовнішніх мережевих затримок і

зосередитися на вимірюванні чистої продуктивності самого серверного застосунку. Тестовий стенд мав таку конфігурацію: процесор AMD Ryzen 5 2600 (6 ядер, 12 потоків), 16 ГБ оперативної пам'яті, операційна система Windows 10.

Сам процес тестування полягав у проведенні серії ідентичних навантажувальних тестів для обох серверів. Оскільки запуск сотень повноцінних клієнтів Unity є непрактичним, для імітації навантаження було розроблено спеціальний скрипт на Node.js. Цей скрипт створював задану кількість WebSocket-клієнтів, кожен з яких після підключення до сервера починав з періодичністю 50 мілісекунд надсилати повідомлення про зміну позиції (POS:), що імітувало активний рух гравця.

Тестування проводилося за єдиним сценарієм: кожна система запускалася з навантаженням у 10, 25, 50, 100, 150 та 200 одночасних клієнтів. Для кожного рівня навантаження система працювала протягом достатнього часу для збору стабільних середніх показників метрик.

- Для Системи А (Node.js), що працювала в Docker-контейнері, навантаження на ЦП та пам'ять фіксувалися за допомогою утиліти docker stats.
- Для Системи Б (C#/.NET), що працювала як нативний процес, ті ж показники фіксувалися за допомогою системного монітора "Диспетчер завдань" (процес GameServerNet.exe).
- Для обох систем середній час відгуку вимірювався безпосередньо в скрипті-симуляторі.

4.4. Аналіз результатів експерименту та порівняння з існуючими рішеннями

Проведене експериментальне тестування дозволило зібрати емпіричні дані про продуктивність двох розроблених серверних прототипів, що є основою для їх об'єктивної порівняльної оцінки. Цей підрозділ присвячений глибокому

аналізу отриманих результатів, виявленню сильних та слабких сторін кожної реалізованої системи .

Для наочності, зведені результати тестування обох систем - Системи А (Node.js) та Системи Б (C#/.NET) - представлені у Таблиці 4.1. Детальна динаміка поведінки кожної системи показана на відповідних графіках нижче.

Таблиця 4.1

Зведені результати порівняльного навантажувального тестування

Кількість гравців	Середня затримка (мс) [Node.js]	Навантаження на ЦП (%) [Node.js, 1 ядро]	Середня затримка (мс) [C#/.NET]	Навантаження на ЦП (%) [C#/.NET, всі ядра]
10	~1.5 - 2.0	~15%	0.62	1.00%
25	~2.5 - 3.0	~40%	1.89	3.00%
50	3.49	100.00%	7.40	11.00%
100	9.56	100.00%	78.45	40.00%
150	~20.0	100.00%	252.80	40.00%
200	30.48	100.00%	731.02	40.00%

Аналіз Системи А (Node.js)

Першим і найважливішим показником є середній час відгуку сервера, оскільки він безпосередньо впливає на якість ігрового досвіду. Залежність цього показника від навантаження наведено на рисунку 4.4.

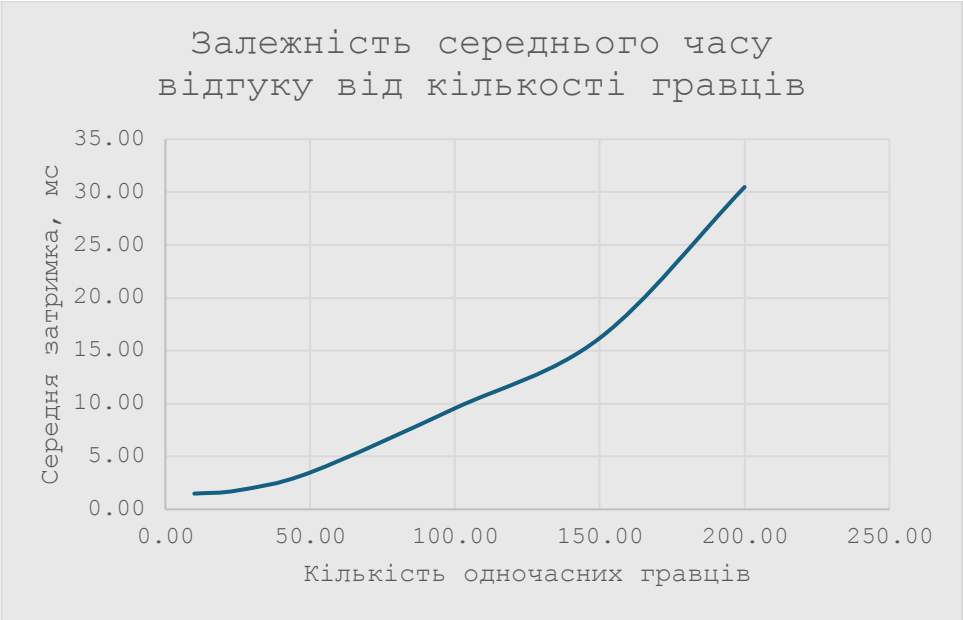

Рисунок 4.4 – Залежність середнього часу відгуку від кількості гравців (Система А: Node.js)

Як видно з графіка, при навантаженні до 50 одночасних гравців система демонструє високу стабільність, а середній час відгуку залишається на дуже низькому рівні, не перевищуючи 3.49 мс . Цей результат є чудовим показником для гри в реальному часі. Однак після досягнення порогу в 50 гравців спостерігається різке, експоненційне зростання затримки: при 100 гравцях вона досягає 9.56 мс, а при 200 гравцях зростає до 30.48 мс. Така динаміка чітко свідчить про те, що система досягла межі своєї продуктивності на одному обчислювальному ядрі, і нові запити починають накопичуватися в черзі.

Для виявлення причини такої деградації було проаналізовано навантаження на центральний процесор, результати якого представлено на рисунку 4.5.

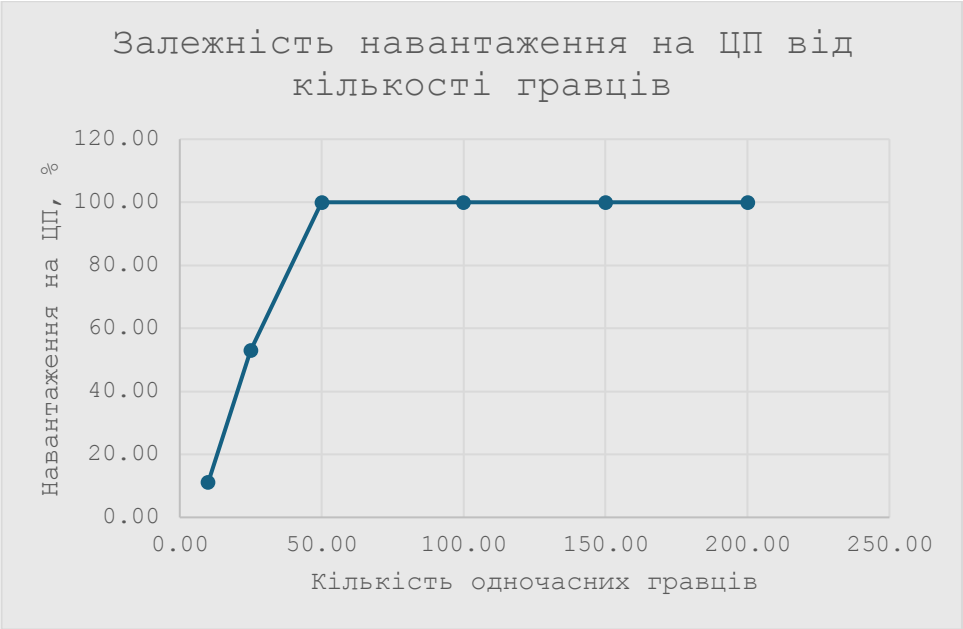

Рисунок 4.5 – Залежність навантаження на ЦП від кількості гравців (Система А: Node.js)

Графік навантаження на ЦП повністю пояснює поведінку системи. Навантаження зростає майже лінійно до позначки в 50 гравців, після чого досягає 100% і виходить на "плато". Це означає, що одне ядро процесора, на якому виконується однопотоковий процес Node.js, виявляється повністю завантаженим. Подальше збільшення кількості гравців не може збільшити навантаження на ЦП, але призводить до формування черги запитів, що й спричиняє стрімке зростання часу відгуку, як показано на рисунку 4.4. Таким чином, експеримент наочно доводить, що саме обчислювальна потужність центрального процесора є головним "вузьким місцем" одного серверного вузла Node.js.

Аналіз Системи Б (C#/.NET)

На відміну від Системи А, результати тестування багатопотокового C#/.NET сервера демонструють кардинально іншу картину. Залежність показників продуктивності для Системи Б наведено на рисунках 4.6 та 4.7.

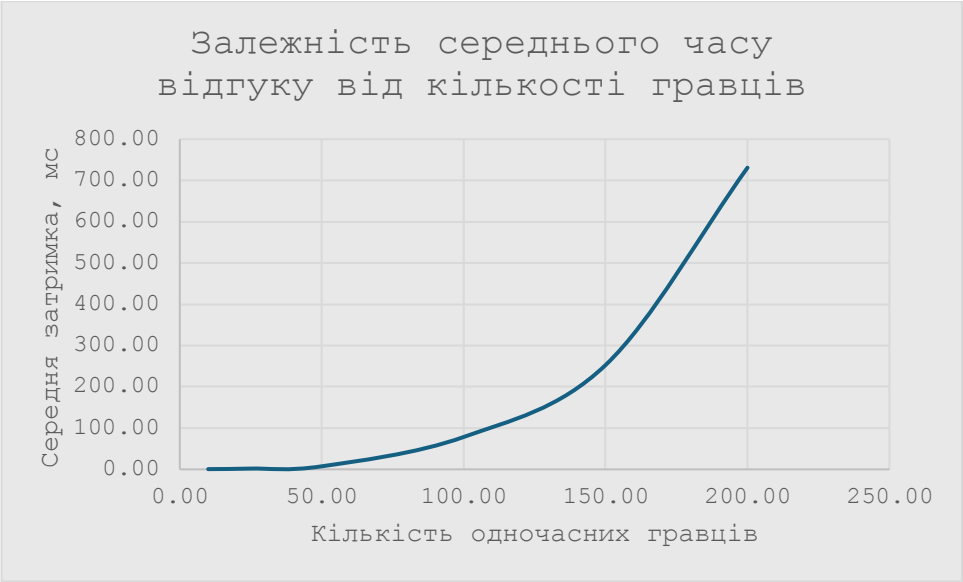

Рисунок 4.6 – Залежність середнього часу відгуку від кількості гравців (Система Б: C#/.NET)

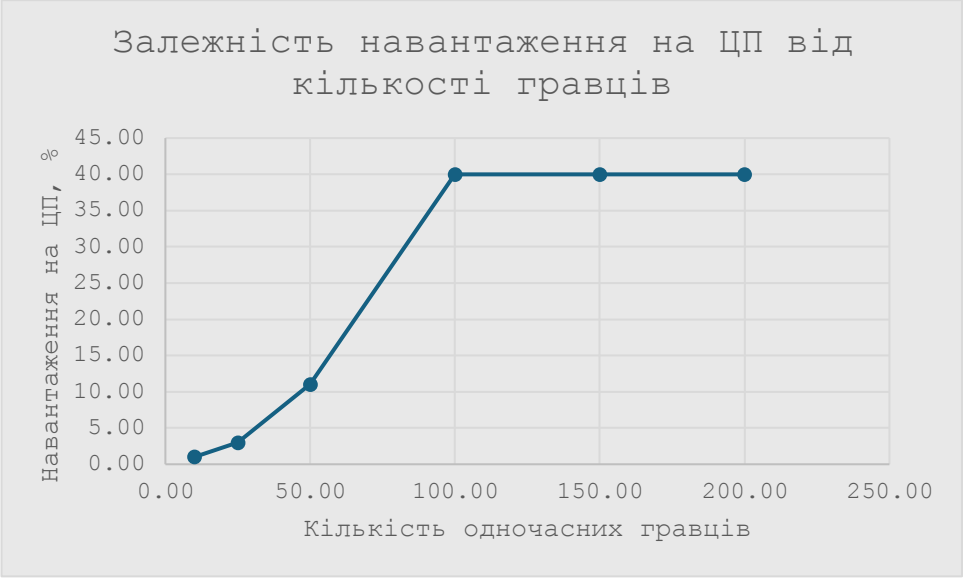

Рисунок 4.7 – Залежність навантаження на ЦП від кількості гравців (Система Б: C#/.NET)

Як видно з Рисунку 4.6, при навантаженні до 50 гравців система C#/.NET демонструє виняткову продуктивність із затримкою, що не перевищує 7.40 мс. Однак, на відміну від плавного зростання в Node.js, тут після 100 гравців спостерігається вибухове, експоненційне зростання затримки - з 78.45 мс до понад 731 мс при 200 гравцях.

Для виявлення причини цієї стрімкої деградації було проаналізовано навантаження на центральний процесор (Рисунок 4.7). Графік показує фундаментальну відмінність від Системи А: навантаження на ЦП зростає лише до 100 гравців, після чого досягає 40% і виходить на "плато".

Порівняльний висновок експерименту

Порівняльний аналіз графіків для Системи А (рис. 4.5) та Системи Б (рис. 4.7) дозволяє зробити ключовий висновок дослідження. Обидві системи демонструють різке зростання затримки, але з фундаментально різних причин:

- Система А (Node.js) виявилася CPU-bound. Її продуктивність жорстко обмежена потужністю одного ядра ЦП, яке було завантажене на 100% .
- Система Б (C#/.NET), навпаки, виявилася I/O-bound (обмеженою операціями вводу-виводу). Її процесор був завантажений лише на 40% і простоював. "Вузьким місцем" стала конкуренція потоків (thread contention) за доступ до спільних ресурсів (ймовірно, мережевих сокетів) під час масової розсилки повідомлень "запит-відповідь". Ця боротьба за ресурси та накладні витрати на синхронізацію призвели до катастрофічного зростання затримки значно раніше, ніж було вичерпано обчислювальні ресурси [35].

Порівняльний аналіз з існуючими рішеннями

При порівнянні з комерційними платформами, такими як AWS GameLift або Photon, розроблене рішення демонструє суттєві переваги у вартості та контролі. Базуючись на технологіях з відкритим кодом, воно має нульову вартість ліцензування, що робить його значно більш економічно привабливим для невеликих команд порівняно з комерційними платформами . Натомість, наш підхід надає повний і необмежений контроль над усіма аспектами серверної логіки. З іншого боку, комерційні платформи, такі як Photon, пропонують готові SDK та повністю абстрагують від розробника всю складність адміністрування інфраструктури. Наш підхід, у свою чергу, вимагає значно глибших знань у сфері DevOps.

У порівнянні з відкритими фреймворками, такими як Colyseus або Mirror, слід відзначити різницю у гнучкості та рівні абстракції. Подібні фреймворки надають готовий каркас для управління сесіями, що значно спрощує процес розробки. Наша реалізація "з нуля" вимагала більших трудовитрат, але натомість дозволила отримати глибоке розуміння всіх внутрішніх процесів та уникнути будь-яких обмежень, що накладаються стороннім фреймворком. Окрім того, важливою є відсутність технологічної прив'язки. Рішення на кшталт Mirror тісно інтегровані виключно з рушієм Unity. Наш сервер, навпаки, є повністю незалежним від клієнта і може взаємодіяти з ігровими клієнтами, розробленими на будь-якому іншому рушії, що робить архітектуру більш універсальною.

4.5. Обґрунтування шляхів оптимізації та подальшого розвитку системи

Проведена реалізація та експериментальне тестування прототипів дозволили не лише підтвердити працездатність спроєктованих архітектур, але й виявити конкретні, фундаментально різні напрямки для їх подальшого вдосконалення. Аналіз результатів, отриманих у попередньому підрозділі, є відправною точкою для визначення шляхів оптимізації, спрямованих на підвищення продуктивності кожної з систем .

Для Системи А (Node.js), "вузьким місцем" якої є 100% завантаження одного ядра ЦП , подальша оптимізація має бути зосереджена на двох напрямках. Фундаментальним рішенням є горизонтальне масштабування, тобто запуск кількох екземплярів Node.js-сервера (наприклад, за допомогою модуля cluster або оркестратора Kubernetes) та балансування навантаження між ними. Додатково, для зниження навантаження на ЦП, можлива оптимізація мережевого протоколу. Наразі для синхронізації використовується повна розсилка об'єкта state; впровадження підходу з надсиланням лише "дельти" змін значно зменшить обсяг даних, що серіалізуються, і, як наслідок, знизить навантаження на процесор.

На відміну від Node.js, тестування Системи Б (C#/.NET) показало, що "вузьким місцем" є не процесор (завантажений лише на 40%), а конкуренція потоків за ресурси вводу-виводу. Це означає, що стратегія оптимізації має бути зовсім іншою. Поточна архітектура "запит-відповідь" є неефективною для багатопотокового середовища. Правильним архітектурним рішенням буде перехід до моделі "Game Loop" - впровадження фонового циклу, який збиратиме всі оновлення та розсилатиме єдиний актуальний стан з фіксованою частотою (наприклад, 20 разів на секунду). Це усуне "шторм" розсилок і вирішить проблему конкуренції потоків. Альтернативою може стати вдосконалення поточної моделі шляхом впровадження асинхронних черг (Async Queues), де потік-обробник лише додає задачу розсилки в чергу, а окремий потік-виконавець її розгрібає, уникаючи блокувань.

Нарешті, для перетворення основного прототипу (Node.js) на повноцінний продукт, необхідно впровадити додаткові сервіси, що забезпечують його життєвий цикл. По-перше, це розширення ігрової логіки, оскільки наразі вороги є статичними; логічним кроком є реалізація штучного інтелекту (AI) для них (наприклад, патрулювання). По-друге, це розширення багатокористувацьких можливостей шляхом розробки кооперативного режиму (лобі). По-третє, це інтеграція систем моніторингу (Prometheus, Grafana) та надійної системи автентифікації, наприклад, на основі токенів (JWT).

ВИСНОВКИ

У ході виконання магістерської дипломної роботи було проведено комплексне дослідження та розробку архітектури розподіленої системи ігрового сервера, спрямованої на вирішення сучасних викликів у сфері багатокористувацьких онлайн-ігор.

На першому етапі було проведено теоретичне дослідження, в ході якого проаналізовано еволюцію архітектурних моделей ігрових серверів, виявлено переваги та недоліки існуючих підходів, а також визначено специфічні вимоги, які висуває до серверної інфраструктури популярний жанр rogelike. Аналіз показав, що традиційні монолітні архітектури є недостатньо гнучкими та масштабованими для обслуговування великої кількості одночасних, ізольованих ігрових сесій.

На основі цього дослідження було спроєктовано та реалізовано два функціонально ідентичних прототипи клієнт-серверної системи для проведення порівняльного аналізу: однопотоковий на базі Node.js та багатопотоковий на базі C#/.NET. Для клієнтської частини в обох випадках використовувався ігровий рушій Unity. Ключовим архітектурним рішенням стало використання технології контейнеризації Docker для забезпечення ізоляції ігрових сесій, що заклало фундамент для подальшого горизонтального масштабування.

Для перевірки життєздатності та порівняльного аналізу продуктивності розроблених прототипів було проведено серію ідентичних експериментальних навантажувальних тестів. Результати тестування виявили фундаментальні відмінності у поведінці систем. Сервер Node.js продемонстрував очікувану поведінку, обмежену потужністю одного ядра ЦП, яке було завантажене на 100% вже при 50-60 гравцях. Натомість, багатопотоковий сервер C#/.NET продемонстрував неочікуваний результат: при стрімкій деградації часу відгуку навантаження на ЦП не перевищувало 40%. Це доводить, що "вузьким місцем"

стала не обчислювальна потужність, а конкуренція потоків (thread contention) за доступ до ресурсів вводу-виводу при масовій розсилці повідомлень.

Таким чином, у ході дослідження було досягнуто поставленої мети. Головним науковим та практичним результатом роботи є кількісне підтвердження того, що вибір серверної архітектури (однопотокова проти багатопотокової) безпосередньо залежить від типу навантаження. Було емпірично доведено, що для задачі типу "запит-відповідь" з масовою розсилкою (I/O-bound), багатопотокова модель може виявитися менш ефективною через накладні витрати на синхронізацію.

Робота доводить, що обрана архітектура не лише вирішує поставлені завдання в рамках прототипу, але й створює чіткі емпіричні обґрунтування для вибору стратегії масштабування (горизонтальне для Node.js або оптимізація I/O для .NET) для побудови надійної, високоефективної розподіленої системи промислового рівня

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What Are Distributed Systems? Електронний ресурс], Режим доступу: https://www.splunk.com/en_us/blog/learn/distributed-systems.html (дата звернення: 04.11.2025).
2. Monoliths vs microservices in gaming architecture: striking the right balance [Електронний ресурс], Режим доступу: https://ascendion.com/insights/monoliths-vs-microservices-in-gaming-architecture-striking-the-right-balance/ (дата звернення: 04.11.2025).
3. BARRI, Ignasi; ROIG, Concepció; GINÉ, Francesc. Distributing game instances in a hybrid client-server/P2P system to support MMORPG playability. Multimedia Tools and Applications, 2016, 75.4: 2005-2029.
4. MELENDEZ, C. Microservices on Kubernetes. DZone, 2019. 7 с. (DZone Refcardz).
5. CAI, W. et al. A Survey on Cloud Gaming: Future of Computer Games. *IEEE Access*. 2016. Vol. 4. P. 7605–7620.
6. Introduction to Scalable Game Development Patterns on AWS [Електронний ресурс], Режим доступу: https://d1.awsstatic.com/whitepapers/aws-scalable-gaming-patterns.pdf (дата звернення: 04.11.2025).
7. Best Practices for Scaling Multiplayer Servers [Електронний ресурс], Режим доступу: https://valngos.com/best-practices-for-scaling-multiplayer-servers.html (дата звернення: 04.11.2025).
8. СЕМЕРЕНКО, В. С. Дослідження та аналіз розробки архітектури ігрових серверів у жанрі «First-person Shooter» : пояснювальна записка до кваліфікаційної роботи магістра. Харків : ХНУРЕ, 2022. 66 с.
9. BALDOVINO, B. An Overview of the Networking Issues of Cloud Gaming: A Literature Review. *Journal of Innovation Information Technology and Application (JINITA)*. 2022. Vol. 4, № 2. P. 120–132.

10. Прискорте сайт з HTTP/3 [Електронний ресурс], Режим доступу: https://hostpro.ua/blog/ua/new-protocol-http3/ (дата звернення: 04.11.2025).

11. FERNANDO, L.; ENGEL, M. M. Comparative Performance Benchmarking of WebSocket Libraries on Node.js and Golang. *Sinkron: Jurnal dan Penelitian Teknik Informatika*. 2025. Vol. 9, № 4. P. 2051–2060.

12. Procedural Map Generation [Електронний ресурс], Режим доступу: https://www.gridsagegames.com/blog/2014/06/procedural-map-generation/ (дата звернення: 04.11.2025).

13. SZABADOS, György N., et al. Roguelike Games - The way we play. International Journal of Engineering and Management Sciences (IJEMS). 2022. Vol. 7, № 4. P. 80–92. DOI: 10.21791/IJEMS.2022.4.7.

14. Amazon GameLift. Що таке Amazon GameLift? [Електронний ресурс], Режим доступу: https://aws.amazon.com/ru/gamelift/ (дата звернення: 04.11.2025).

15. Photon Engine - Realtime Multiplayer Game Development [Електронний ресурс], Режим доступу: https://doc.photonengine.com/fusion/current/fusion-2-intro (дата звернення: 04.11.2025).

16. AGONES. Agones: Dedicated Game Server Hosting on Kubernetes [Електронний ресурс], Режим доступу: https://agones.dev/site/docs/ (дата звернення: 04.11.2025).

17. Colyseus: Multiplayer Game Server for Node.js [Електронний ресурс], Режим доступу: https://docs.colyseus.io/ (дата звернення: 04.11.2025).

18. Mirror: Networking for Unity [Електронний ресурс], Режим доступу: https://mirror-networking.gitbook.io/docs/ (дата звернення: 04.11.2025).

19. .NET vs Node.js: What to Choose in 2025 [Електронний ресурс], Режим доступу: https://www.techmagic.co/blog/node-js-vs-net-what-to-choose (дата звернення: 04.11.2025).

20. ASP.NET Core vs Node.js: Which Backend Framework Reigns Supreme in 2024? [Електронний ресурс], Режим доступу:

https://medium.com/@solomongetachew112/asp-net-core-vs-node-js-which-backend-framework-reigns-supreme-in-2024-c77f6a26d81a (дата звернення: 04.11.2025).

21. UNITY. Netcode for GameObjects: Networking library for Unity [Електронний ресурс], Режим доступу: https://unity.com/products/netcode (дата звернення: 04.11.2025).

22. Docker - Офіційна документація [Електронний ресурс], Режим доступу: https://docs.docker.com/manuals/ (дата звернення: 04.11.2025).

23. VU, V. H. Deployment and Hosting of an Online Game Server: Case: Google Cloud Platform : Bachelor's thesis. Turku : Turku University of Applied Sciences, 2021. 49 с.

24. Kubernetes - Офіційна документація [Електронний ресурс], Режим доступу: https://kubernetes.io/docs/home/ (дата звернення: 04.11.2025).

25. HOHN, A.; GUPTA, A. Getting Started with Kubernetes. DZone, 2018. 8 с. (DZone Refcardz).

26. Docker and Kubernetes Deployment for Game Server Hosting [Електронний ресурс], Режим доступу: https://www.varidata.com/blog-en/docker-and-kubernetes-deployment-for-game-server-hosting/ (дата звернення: 04.11.2025).

27. Scaling Dedicated Game Servers with Kubernetes: Part 1 – Containerising and Deploying [Електронний ресурс], Режим доступу: https://www.gamedeveloper.com/programming/scaling-dedicated-game-servers-with-kubernetes-part-1-containerising-and-deploying?utm_source=chatgpt.com (дата звернення: 04.11.2025).

28. How do load balancer algorithms distribute client traffic across servers? [Електронний ресурс], Режим доступу: https://kemptechnologies.com/load-balancer/load-balancing-algorithms-techniques (дата звернення: 04.11.2025).

29. Node.js - Офіційна документація [Електронний ресурс], Режим доступу: https://nodejs.org/docs/latest/api/ (дата звернення: 04.11.2025).

30. Unity - Офіційна документація [Електронний ресурс], Режим доступу: https://docs.unity3d.com/6000.2/Documentation/Manual/UnityManual.html (дата звернення: 04.11.2025).

31. Miell I., Sayers A. Docker in Practice. 2-ге вид. Shelter Island : Manning Publications, 2019. 384 с.

32. MongoDB - Офіційна документація [Електронний ресурс], Режим доступу: https://www.mongodb.com/docs/ (дата звернення: 04.11.2025).

33. Intro to Kubernetes Autoscaling and Best Practices for Implementations [Електронний ресурс], Режим доступу: https://stormforge.io/kubernetes-autoscaling/ (дата звернення: 04.11.2025).

34. Бібліотека ws: a Node.js WebSocket library / npm [Електронний ресурс], Режим доступу: https://www.npmjs.com/package/ws (дата звернення: 04.11.2025).

35. MICROSOFT. Asynchronous programming with async and await [Електронний ресурс], Режим доступу: https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/ (дата звернення: 04.11.2025).

ДОДАТКИ

Додаток А

Код серверної частини на Node.js

```
import express from "express";
import { WebSocketServer } from "ws";
import crypto from "crypto";
import { WebSocket } from 'ws';

const app = express();
const PORT = 8080;

const server = app.listen(PORT, () => {
  console.log(`· HTTP + WS server running on port ${PORT}`);
});

const wss = new WebSocketServer({ server });

// ------------------ Налаштування карти ------------------
const MAP_SIZE = 40;
const ROOM_COUNT = 6;
const ROOM_MIN_SIZE = 5;
const ROOM_MAX_SIZE = 9;
const TREASURE_COUNT = 10;

let players = {};
const clients = new Map();
let mapData = generateSewerMap();

// ------------------ Головна логіка ------------------
wss.on("connection", (ws) => {
  const clientId = crypto.randomUUID();

  const spawn = mapData.startPosition;
  // ✅ Додаємо поле score для кожного гравця при підключенні
  players[clientId] = { x: spawn.x, y: 1, z: spawn.z, score: 0 };

  console.log(`✅ Client connected: ${clientId}`);
  clients.set(clientId, ws);
  ws.clientId = clientId;

  ws.send(JSON.stringify({
    type: "welcome",
    clientId,
    startPosition: spawn
  }));

  ws.send(JSON.stringify({
    type: "map",
    blocks: mapData.blocks,
    treasures: mapData.treasures,
    enemies: mapData.enemies,
    startMarker: mapData.startMarker,
    endMarker: mapData.endMarker
  }));

  ws.on("message", (msg) => {
    const text = msg.toString();
```

```
            if (text.startsWith("POS:")) {
              const [x, y, z] = text.split(":")[1].split(",").map(Number);

              if (!isNaN(x) && !isNaN(y) && !isNaN(z)) {
                  // Зберігаємо координати, але також і рахунок, щоб не затерти його
                  if(players[clientId]){
                     const currentScore = players[clientId].score || 0;
                     players[clientId] = { x, y, z, score: currentScore };
                  }
              }

              broadcastState();

            } else if (text.startsWith("TAKE_TREASURE:")) {
                //  логіка для обробки підбору скарбу
                try {
                    const posJson = text.substring("TAKE_TREASURE:".length);
                    const pos = JSON.parse(posJson);

                    // Шукаємо скарб на карті за координатами
                    const treasureIndex = mapData.treasures.findIndex(t => t.x === pos.x && t.z ===
pos.z);

                    if (treasureIndex !== -1) {
                        // Знайшли! Видаляємо його з масиву
                        mapData.treasures.splice(treasureIndex, 1);

                        // Нараховуємо очко гравцю
                        if (players[clientId]) {
                            players[clientId].score++;
                        }

                         console.log(`· Treasure at (${pos.x}, ${pos.z}) taken by ${clientId}. New
score: ${players[clientId]?.score}`);

                        // Розсилаємо ВСІМ, що скарб зник
                        const treasureGoneMsg = JSON.stringify({
                            type: "TREASURE_GONE",
                            position: pos
                        });
                        wss.clients.forEach(client => {
                            if (client.readyState === 1) client.send(treasureGoneMsg);
                        });

                        // Оновлюємо стан гравців (щоб передати новий рахунок)
                        broadcastState(true);
                    }
                } catch (e) { console.error("Failed to parse TAKE_TREASURE", e); }

            }
            else if (text === "HIT_ENEMY") {
                const player = players[clientId];
                if (player) {
                    console.log(`- Player ${clientId} hit an enemy. Resetting...`);

                    // Обнуляємо рахунок
                    player.score = 0;
```

```
            // Повертаємо гравця на стартову позицію поточної карти
            player.x = mapData.startPosition.x;
            player.z = mapData.startPosition.z;

            // Надсилаємо оновлений стан усім гравцям
            broadcastState(true);
          }
        } else if (text === "FINISH") {
          console.log(`· FINISH received from ${clientId}. Regenerating map...`);

          mapData = generateSewerMap();

          const newMapMessage = JSON.stringify({
            type: "map",
            blocks: mapData.blocks,
            treasures: mapData.treasures,
            enemies: mapData.enemies,
            startMarker: mapData.startMarker,
            endMarker: mapData.endMarker
          });

          const newSpawn = mapData.startPosition;
          for (const id in players) {
            // ✅ При рестарті зберігаємо рахунок
            const old_score = players[id].score || 0;
            players[id] = { x: newSpawn.x, y: 1, z: newSpawn.z, score: old_score };
          }

          wss.clients.forEach((client) => {
            if (client.readyState === 1) {
              client.send(newMapMessage);
            }
          });

          broadcastState(true);
        }
      });

      ws.on("close", () => {
        delete players[clientId];
        clients.delete(clientId);
        console.log(`❌ Client disconnected: ${clientId}`);
        broadcastState();
      });
    });

    function broadcastState(all = false) {
      clients.forEach((ws, clientId) => {
        if (ws.readyState === WebSocket.OPEN) {
          const otherPlayers = Object.keys(players)
            .filter(key => key !== clientId || all)
            .reduce((obj, key) => {
              obj[key] = players[key];
              return obj;
            }, {});

          ws.send(JSON.stringify({
            type: 'state',
```

```
        players: otherPlayers,
      }));
    }
  });
}

// ------------------ Генерація карти ------------------
function generateSewerMap() {
  const blocks = Array.from({ length: MAP_SIZE }, () => Array(MAP_SIZE).fill(1));
  let rooms = [];

  for (let i = 0; i < ROOM_COUNT; i++) {
    const w = rand(ROOM_MIN_SIZE, ROOM_MAX_SIZE);
    const h = rand(ROOM_MIN_SIZE, ROOM_MAX_SIZE);
    const x = rand(1, MAP_SIZE - w - 1);
    const y = rand(1, MAP_SIZE - h - 1);

    rooms.push({ x, y, w, h });
    for (let r = x; r < x + w; r++) {
      for (let c = y; c < y + h; c++) {
        blocks[r][c] = 0;
      }
    }
  }

  for (let i = 1; i < rooms.length; i++) {
    const prev = rooms[i - 1];
    const curr = rooms[i];
    const x1 = Math.floor(prev.x + prev.w / 2);
    const y1 = Math.floor(prev.y + prev.h / 2);
    const x2 = Math.floor(curr.x + curr.w / 2);
    const y2 =  Math.floor(curr.y + curr.h / 2);
    carvePath(blocks, x1, y1, x2, y2);
  }

  const startRoom = rooms[0];
  const endRoom = rooms[rooms.length - 1];
  const startPos = {
    x: startRoom.x + Math.floor(startRoom.w / 2),
    z: startRoom.y + Math.floor(startRoom.h / 2)
  };
  const endPos = {
    x: endRoom.x + Math.floor(endRoom.w / 2),
    z: endRoom.y + Math.floor(endRoom.h / 2)
  };

  if (blocks[startPos.x]?.[startPos.z] !== undefined)
    blocks[startPos.x][startPos.z] = 0;

  if (blocks[endPos.x]?.[endPos.z] !== undefined)
    blocks[endPos.x][endPos.z] = 0;

  const startMarker = { x: startPos.x, z: startPos.z };
  const endMarker = { x: endPos.x, z: endPos.z };

  const enemies = rooms
    .filter((room, i) => i !== 0 && Math.random() < 0.6)
    .map(room => {
      const enemyPosition = {
```

```
        x: room.x + Math.floor(room.w / 2),
        z: room.y + Math.floor(room.h / 2)
      };
      if (room === endRoom) {
        enemyPosition.x += 1;
      }
      return enemyPosition;
    });

  console.log(`--- Map Generated: ${enemies.length} enemies found. ---`);

  const treasures = [];
  let placed = 0;
  while (placed < TREASURE_COUNT) {
    const x = rand(2, MAP_SIZE - 3);
    const z = rand(2, MAP_SIZE - 3);
    if (blocks[x]?.[z] === 0) {
      if ((x !== startPos.x || z !== startPos.z) && (x !== endPos.x || z !== endPos.z)) {
        treasures.push({ x, z });
        placed++;
      }
    }
  }

  return { blocks, treasures, enemies, startPosition: startPos, startMarker, endMarker };
}

function rand(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}

function carvePath(blocks, x1, y1, x2, y2) {
  let x = x1;
  let y = y1;
  while (x !== x2) {
    if (blocks[x] !== undefined) blocks[x][y] = 0;
    x += x < x2 ? 1 : -1;
  }
  while (y !== y2) {
    if (blocks[x] !== undefined) blocks[x][y] = 0;
    y += y < y2 ? 1 : -1;
  }
}
```

Додаток Б

Код ключових скриптів клієнта на Unity

GameClient

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;
using NativeWebSocket;
using System.Text;
using Newtonsoft.Json.Linq;
using System.Globalization;
using TMPro;

public class GameClient : MonoBehaviour
{
    private WebSocket websocket;
    private CharacterController controller;
    private Vector2 moveInput;
    public float speed = 5f;

    public float gravity = -9.81f;
    private float velocityY = 0f;

    public GameObject playerPrefab;
       private Dictionary<string, GameObject> otherPlayers = new Dictionary<string,
GameObject>();
    private string clientId;

    private bool hasSpawned = false;

    public event System.Action<string> OnServerMessage;
```

```
    public MapAndCamera mapAndCamera;
    public TextMeshProUGUI scoreText;

    private int score = 0; // ✅ Зберігаємо рахунок локально

    async void Start()
    {
        controller = GetComponent<CharacterController>();
        websocket = new WebSocket("ws://localhost:5232");

        websocket.OnOpen += () => { Debug.Log("✅ Connected to WebSocket server"); };

        websocket.OnMessage += (bytes) =>
        {
            string message = Encoding.UTF8.GetString(bytes);
            // Викликаємо обробник в основному потоці Unity
                                    UnityMainThreadDispatcher.Instance().Enqueue(()     =>
HandleServerMessage(message));
            OnServerMessage?.Invoke(message);
        };

        websocket.OnError += (err) => { Debug.LogError("❌ WebSocket Error: " + err); };
        websocket.OnClose += (e) => { Debug.Log("· WebSocket connection closed"); };

        await websocket.Connect();
    }

    void Update()
    {
        websocket?.DispatchMessageQueue();

        if (!hasSpawned) return;
        if (controller == null || !controller.enabled) return;

        Vector3 move = new Vector3(moveInput.x, 0, moveInput.y) * speed;

        if (controller.isGrounded && velocityY < 0)
            velocityY = -1f;
        else
            velocityY += gravity * Time.deltaTime;

        Vector3 velocity = new Vector3(move.x, velocityY, move.z);
        controller.Move(velocity * Time.deltaTime);

        // Надсилаємо позицію, тільки якщо є рух
        if (move.magnitude > 0.01f && websocket.State == WebSocketState.Open)
        {
            string pos = string.Format(CultureInfo.InvariantCulture, "POS:{0},{1},{2}",
                transform.position.x, transform.position.y, transform.position.z);
            websocket.SendText(pos);
        }
    }

    public void OnMove(InputAction.CallbackContext context)
    {
        moveInput = context.ReadValue<Vector2>();
    }


    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Finish"))
        {
            if (websocket != null && websocket.State == WebSocketState.Open)
            {
                websocket.SendText("FINISH");
            }
        }

        else if (other.CompareTag("Treasure"))
        {
            if (websocket != null && websocket.State == WebSocketState.Open)
            {
```

```
                        var treasure = other.GetComponent<Treasure>();
                        if (treasure != null)
                        {
                            // Створюємо JSON з координатами скарбу
                                                                                string      posJson      =
$"{{\"x\":{treasure.position.x},\"z\":{treasure.position.z}}}";
                            // Надсилаємо повідомлення на сервер
                            websocket.SendText("TAKE_TREASURE:" + posJson);

                            // Миттєво ховаємо скарб, щоб не було затримки для гравця
                            other.gameObject.SetActive(false);
                        }
                    }
                }
                else if (other.CompareTag("Enemy"))
            {
                Debug.Log("Hit an enemy! Sending message to server.");
                if (websocket != null && websocket.State == WebSocketState.Open)
                {
                    // Надсилаємо серверу повідомлення, що ми торкнулися ворога
                    websocket.SendText("HIT_ENEMY");
                }
            }
            }


            private void HandleServerMessage(string msg)
            {
                JObject json;
                try { json = JObject.Parse(msg); }
                catch { return; }
                string type = (string)json["type"];

                if (type == "welcome")
                {
                    clientId = (string)json["clientId"];
                    var start = json["startPosition"];
                    if (start != null)
                    {
                        float x = start["x"].ToObject<float>();
                        float z = start["z"].ToObject<float>();


                        if (controller != null)
                        {
                            controller.enabled = false;
                            transform.position = new Vector3(x, 1.5f, z);
                            controller.enabled = true;
                        }

                        hasSpawned = true;
                    }
                }
                else if (type == "state")
                {
                    var playersObj = json["players"] as JObject;
                    if (playersObj == null) return;

                    foreach (var prop in playersObj.Properties())
                    {
                        string id = prop.Name;
                        var p = prop.Value;

                        if (p?["x"] == null || p?["y"] == null || p?["z"] == null) continue;

                        float x = p["x"].ToObject<float>();
                        float y = p["y"].ToObject<float>();
                        float z = p["z"].ToObject<float>();

                        if (id == clientId)
                        {
                            // ✅ Оновлюємо рахунок, коли приходить стан
                            if (p["score"] != null)
                            {
                                score = p["score"].ToObject<int>();
                                if (scoreText != null)
```

```
                            {
                                scoreText.text = "Скарби: " + score;
                            }
                        }

                        Vector3 serverPos = new Vector3(x, 1.5f, z);

                        if (Vector3.Distance(transform.position, serverPos) > 1.5f)
                        {
                            if (controller != null)
                            {
                                controller.enabled = false;
                                transform.position = serverPos;
                                controller.enabled = true;
                            }
                        }
                    }
                    else // Обробляємо інших гравців
                    {
                        Vector3 serverPos = new Vector3(x, y, z);
                        if (!otherPlayers.ContainsKey(id))
                        {
                            if (playerPrefab != null)
                            {
                                        GameObject go = Instantiate(playerPrefab, serverPos,
Quaternion.identity);
                                go.name = "Player_" + id.Substring(0, 6);
                                otherPlayers[id] = go;
                            }
                        }
                        else
                        {
                            if(otherPlayers[id] != null)
                                                    otherPlayers[id].transform.position =
Vector3.Lerp(otherPlayers[id].transform.position, serverPos, Time.deltaTime * 10f);
                        }
                    }
                }

                    // Логіка видалення гравців, що відключилися
                    List<string> toRemove = new List<string>();
                    foreach (var kv in otherPlayers)
                    {
                        if (!playersObj.ContainsKey(kv.Key))
                        {
                            if (kv.Value != null) Destroy(kv.Value);
                            toRemove.Add(kv.Key);
                        }
                    }
                    foreach (var id in toRemove)
                        otherPlayers.Remove(id);
                }

                else if (type == "TREASURE_GONE")
                {
                    var pos = json["position"];
                    if (pos != null)
                    {
                        float x = pos["x"].ToObject<float>();
                        float z = pos["z"].ToObject<float>();

                       Debug.Log($"[!!!] Отримано команду видалити скарб на позиції: ({x}, {z})");
                        // Викликаємо метод на генераторі карти, щоб він видалив об'єкт
                        if (mapAndCamera != null)
                        {
                            mapAndCamera.RemoveTreasure(x, z);
                        }
                        else
                    {
                    // ✅ Додаткова перевірка, якщо посилання на генератор порожнє
                      Debug.LogError("[!!!] Не можу видалити скарб, бо посилання на MapAndCamera
порожнє!");
                    }

                    }
                }
```

```
        }

        private async void OnApplicationQuit()
        {
            if (websocket != null)
                await websocket.Close();
        }
    }
```

MapAndCamera

```
    using UnityEngine;
    using Newtonsoft.Json.Linq;
    using System.Collections.Generic;

    public class MapAndCamera : MonoBehaviour
    {
        public Camera mainCamera;
        public Transform playerTarget;


        public GameObject treasurePrefab;

        private List<GameObject> mapObjects = new List<GameObject>();

          private Dictionary<string, GameObject> spawnedTreasures = new Dictionary<string,
GameObject>();
        private List<GameObject> enemyObjects = new List<GameObject>();
        private List<GameObject> markerObjects = new List<GameObject>();
        private GameClient gameClient;
        public GameObject enemyPrefab;

        void Start()
        {
            gameClient = FindFirstObjectByType<GameClient>();
            if (gameClient != null)
            {
                gameClient.OnServerMessage += HandleServerMessage;
            }
            else
            {
                Debug.LogError("MapAndCamera script could not find the GameClient script on the
scene!");
            }
        }

        void LateUpdate()
        {
            if (playerTarget != null && mainCamera != null)
            {
                Vector3 camPos = playerTarget.position + new Vector3(0, 12, -12);
                  mainCamera.transform.position = Vector3.Lerp(mainCamera.transform.position,
camPos, Time.deltaTime * 5);
                mainCamera.transform.LookAt(playerTarget);
            }
        }

        public void HandleServerMessage(string msg)
        {
            JObject json;
            try { json = JObject.Parse(msg); }
            catch { return; }
            string type = (string)json["type"];

            if (type == "map")
            {
                // Clear all old objects
                foreach (var obj in mapObjects) Destroy(obj);
                mapObjects.Clear();

                // ✅ Очищуємо скарби логікою
                foreach (var kvp in spawnedTreasures) Destroy(kvp.Value);
                spawnedTreasures.Clear();
```

```
                    foreach (var obj in enemyObjects) Destroy(obj);
                    enemyObjects.Clear();
                    foreach (var obj in markerObjects) Destroy(obj);
                    markerObjects.Clear();

                    DrawMap(json["blocks"] as JArray);
                    DrawTreasures(json["treasures"] as JArray);
                    DrawEnemies(json["enemies"] as JArray);


                    if (json["startMarker"] != null)
                    {
                        int sx = (int)json["startMarker"]["x"];
                        int sz = (int)json["startMarker"]["z"];
                        GameObject startCube = GameObject.CreatePrimitive(PrimitiveType.Cube);
                        startCube.name = "Start_Marker";
                        startCube.transform.position = new Vector3(sx, 0.2f, sz);
                        startCube.transform.localScale = new Vector3(1f, 0.2f, 1f);
                        startCube.GetComponent<Renderer>().material.color = Color.green;
                        Destroy(startCube.GetComponent<Collider>());
                        markerObjects.Add(startCube);
                    }

                    if (json["endMarker"] != null)
                    {
                        int ex = (int)json["endMarker"]["x"];
                        int ez = (int)json["endMarker"]["z"];

                                                            GameObject      visibleFinishCube      =
GameObject.CreatePrimitive(PrimitiveType.Cube);
                        visibleFinishCube.name = "Finish_Visual_Cube";
                        visibleFinishCube.transform.position = new Vector3(ex, 0.5f, ez);
                        visibleFinishCube.transform.localScale = new Vector3(0.8f, 0.8f, 0.8f);
                        visibleFinishCube.GetComponent<Renderer>().material.color = Color.cyan;
                        Destroy(visibleFinishCube.GetComponent<Collider>());
                        markerObjects.Add(visibleFinishCube);

                                                          GameObject     invisibleFinishTrigger     =
GameObject.CreatePrimitive(PrimitiveType.Cube);
                        invisibleFinishTrigger.name = "Finish_Invisible_Trigger";
                        invisibleFinishTrigger.transform.position = new Vector3(ex, 1.5f, ez);
                        invisibleFinishTrigger.transform.localScale = new Vector3(1f, 3f, 1f);
                        invisibleFinishTrigger.GetComponent<MeshRenderer>().enabled = false;
                        invisibleFinishTrigger.GetComponent<Collider>().isTrigger = true;
                        invisibleFinishTrigger.tag = "Finish";
                        markerObjects.Add(invisibleFinishTrigger);
                    }
                }
            }

            private void DrawMap(JArray blockRows)
            {

                for (int r = 0; r < blockRows.Count; r++)
                {
                    var cols = blockRows[r] as JArray;
                    for (int c = 0; c < cols.Count; c++)
                    {
                        int value = (int)cols[c];
                        GameObject floor = GameObject.CreatePrimitive(PrimitiveType.Cube);
                        floor.transform.position = new Vector3(r, 0, c);
                        floor.transform.localScale = new Vector3(1, 0.2f, 1);
                          floor.GetComponent<Renderer>().material.color = new Color(0.35f, 0.35f,
0.35f);
                        mapObjects.Add(floor);

                        if (value > 0)
                        {
                            GameObject block = GameObject.CreatePrimitive(PrimitiveType.Cube);
                            block.transform.position = new Vector3(r, 0.6f, c);
                            block.GetComponent<Renderer>().material.color = Color.gray;
                            mapObjects.Add(block);
                        }
                    }
                }
            }
```

```
            private void DrawTreasures(JArray treasures)
            {
                if (treasurePrefab == null) {
                    Debug.LogError("Treasure Prefab is not assigned in the MapAndCamera script!");
                    return;
                }

                foreach (var t in treasures)
                {
                    float x = t["x"].ToObject<float>();
                    float z = t["z"].ToObject<float>();
                    Vector3 pos = new Vector3(x, 0.5f, z);


                     GameObject treasureGO = Instantiate(treasurePrefab, pos, Quaternion.identity,
transform);


                    treasureGO.GetComponent<Treasure>().position = pos;

                    // Зберігаємо скарб у словнику за ключем "x,z" для подальшого доступу
                    string key = $"{x},{z}";
                    spawnedTreasures[key] = treasureGO;
                }
            }


            public void RemoveTreasure(float x, float z)
            {
                string key = $"{x},{z}";
                if (spawnedTreasures.ContainsKey(key))
                {

                    if(spawnedTreasures[key] != null)
                    {
                        Destroy(spawnedTreasures[key]);
                    }
                    // Видаляємо запис зі словника
                    spawnedTreasures.Remove(key);
                }
            }

            private void DrawEnemies(JArray enemies)
        {
            if (enemyPrefab == null) {
                 Debug.LogError("Enemy Prefab is not assigned in the MapAndCamera script!");
                 return;
            }

            foreach (var e in enemies)
            {
                int x = (int)e["x"];
                int z = (int)e["z"];
                // Створюємо ворога з префабу
                     GameObject  enemyGO  =  Instantiate(enemyPrefab,  new  Vector3(x,  0.5f,  z),
Quaternion.identity, transform);

                enemyObjects.Add(enemyGO);
            }
        }

            private void OnDestroy()
            {
                if (gameClient != null)
                    gameClient.OnServerMessage -= HandleServerMessage;
            }
        }
```

UnityMainThreadDispatcher

```
        using System;
        using System.Collections.Generic;
```

```
using UnityEngine;

/// <summary>
/// A thread-safe class which holds a queue of actions to execute on the next Update() method. It can be used to make calls to the main thread for things such as UI Manipulation in Unity. It was developed for use in combination with the Firebase Unity plugin, which uses separate threads for event handling
/// </summary>
public class UnityMainThreadDispatcher : MonoBehaviour
{
    private static readonly Queue<Action> _executionQueue = new Queue<Action>();

    public void Update()
    {
        lock (_executionQueue)
        {
            while (_executionQueue.Count > 0)
            {
                _executionQueue.Dequeue().Invoke();
            }
        }
    }

    /// <summary>
    /// Locks the queue and adds the Action to the queue
    /// </summary>
    /// <param name="action">The action to execute</param>
    public void Enqueue(Action action)
    {
        lock (_executionQueue)
        {
            _executionQueue.Enqueue(action);
        }
    }

    private static UnityMainThreadDispatcher _instance = null;

    public static bool Exists()
    {
        return _instance != null;
    }

    public static UnityMainThreadDispatcher Instance()
    {
        if (!Exists())
        {
                    throw new Exception("UnityMainThreadDispatcher could not find the UnityMainThreadDispatcher object. Please ensure you have added the MainThreadExecutor Prefab to your scene.");
        }
        return _instance;
    }

    void Awake()
    {
        if (_instance == null)
        {
            _instance = this;
            DontDestroyOnLoad(this.gameObject);
```

```
        }
    }

    void OnDestroy()
    {
        _instance = null;
    }
}
```