Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Методи та алгоритми організації мережевого програмування мультиплеєрної системи ігрового застосунку
(тема)

Виконав:
здобувач 2 року навчання,
групи СПм-23-3
Олексій ДОЛГОПОЛОВ
(власне ім’я, прізвище)

Спеціальність
123 «Комп’ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма
Системне програмування
(повна назва освітньої програми)

Керівник: проф. Тетяна ФЕСЕНКО
(посада, власне ім’я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ ________ Андрій КОВАЛЕНКО
(підпис) (власне ім’я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

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

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________________
(підпис)

"____"________________ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Долгополову Олексію Максимовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Методи та алгоритми організації мережевого програмування мультиплеєрної системи ігрового застосунку

затверджена наказом по університету від " 21 " квітня 2025 р. № 296 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 16 червня 2025 р.

3. Вхідні дані до роботи 1) стек протоколів: TCP/IP, Client-server, Request/response, Broadcast, Ping; 2) Id, Packet, Items, NPCs ; 3) C# Unity.

4. Перелік питань, що потрібно опрацювати у роботі 1) проведення аналізу існуючих рішень, що реалізують мережеву взаємодію у ігрових додатках;
2) вибір технологій основуючись на проведеному аналізі;
3) створення власної реалізації моделі;
4) тестування моделей та аналіз результатів.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп’ютерних ілюстрацій

Слайд-презентація – 11 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім’я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Постановка задачі	21.04.2025-26.04.2025	
2	Проєктування системи	26.04.2025-30.04.2025	
3	Реалізація	30.04.2025-20.05.2025	
4	Тестування і експериментальна перевірка	20.05.2025-23.05.2025	
5	Аналіз результатів	23.05.2025-26.05.2025	
6	Висновки та перспективи	26.05.2025-27.05.2025	
7	Оформлення матеріалів кваліфікаційної роботи	27.05.2025-02.06.2025	
8	Подання кваліфікаційної роботи керівникові	02.06.2025-10.06.2025	
	та її попередній захист		
9	Подання кваліфікаційної роботи на	10.06.2025-12.06.2025	
	рецензування		

Дата видачі завдання “ 21 ” квітня 20 25 р.

Здобувач
(підпис)

Керівник роботи ______________ проф. Тетяна ФЕСЕНКО
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 97 с., 30 рис., 5 табл., 1 дод., 20 джерел.

КОМП’ЮТЕРНА МЕРЕЖА, ІНТЕРНЕТ, ПАКЕТ, ПРОТОКОЛ, ШВИДКІСТЬ, TCP, СИНХРОНІЗАЦІЯ, UNITY, ІГРОВИЙ ЗАСТОСУНОК.

Метою кваліфікаційної роботи є розробка моделі та методів мережевої частини для ігрових застосунків. Основною причиною розробки стало, те що поточні рішення не повністю покривають потреби розробників у швидкості передачі даних, що може призвести до негативного досвіду і кінцевого користувача. Також значною частиною є те, що поточні рішення не надають великою масштабованості та інструментів для вдосконалення існуючих продуктів, де заздалегідь доволі закритий функціонал.

У ході виконання кваліфікаційної роботи було проведено аналіз існуючих рішень мережевих систем у ігрових додатках, що дозволило сформулювати та встановити напрямок у подальшій розробці власного рішення.

Було розроблено модель мережевої частини, де основний напрямок розробки був прискорення передачі пакетів з даними у сучасних ігрових застосунках. Розроблена модель є гнучким рішенням, яке дозволяє розробникам легко налаштовувати розмір і як будуть передаватися пакет з потрібними даними, їх мету та пріоритет.

ABSTRACT

Master's thesis: 97 pages, 30 figures, 5 tables, 1 appendices, 20 sources.

COMPUTER NETWORK, INTERNET, PACKET, PROTOCOL, SPEED, TCP, SYNCHRONIZATION, UNITY, GAME APPLICATION.

The major goal of this thesis is to develop a model and methods for the networking component of game applications. The main reason for this development is that current solutions do not fully meet the needs of developers in terms of data transmission speed, which can lead to a negative experience for the end user. Another significant factor is that current solutions lack scalability and tools for improving existing products, where the functionality is often quite closed and rigid from the start.

In order to define the direction for creating a custom solution, this thesis includes an analysis of existing networking systems in game applications, which helped to identify key limitations and requirements.

A networking model was developed, with the primary focus on accelerating packet transmission in modern game applications. The resulting model is a flexible solution that allows developers to easily configure the size, transmission method, purpose, and priority of data packets.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ ... 8
ВСТУП ... 9
1 СУЧАСНІ ТЕХНОЛОГІЇ ОРГАНІЗАЦІЇ МЕРЕЖЕВОГО ПРОГРАМУВАННЯ МУЛЬТИПЛЕЄРНИХ СИСТЕМ ... 11
1.1 Аналіз існуючих рішень для створення мультиплеєрної системи ... 11
1.1.1 UNet (Unity Networking) ... 11
1.1.2 Photon ... 12
1.1.3 Mirror ... 14
1.1.4 Netcode for GameObjects ... 15
1.2 Класифікація мультиплеєрних архітектур ... 17
1.2.1 Клієнт-серверна архітектура ... 17
1.2.2 Peer-to-peer архітектура ... 19
1.2.3 Гібридні архітектури ... 21
1.2.4 Архітектура з розподіленим авторитетом ... 22
1.2.5 Архітектури для масштабних онлайн-ігор ... 23
1.2.6 Критерії вибору архітектури для конкретного проекту ... 25
1.3 Протоколи передачі даних у мультиплеєрних системах ... 26
1.3.1 Основи мережевої взаємодії в мультиплеєрних системах ... 26
1.3.2 TCP (Transmission Control Protocol) ... 28
1.3.3 UDP (User Datagram Protocol) ... 30
1.3.4 Гібридні рішення на основі TCP/UDP ... 31
1.4 Постановка завдань дослідження ... 32
2 МЕТОДИ ОРГАНІЗАЦІЇ МЕРЕЖЕВОЇ ВЗАЄМОДІЇ В ІГРОВИХ ЗАСТОСУНКАХ ... 34
2.1 Архітектура мережевого коду ... 34
2.2 Методи забезпечення синхронізації та відмовостійкості ... 37
2.3 Алгоритми оптимізації мережевого трафіку ... 41

2.4 Виявлення та обробка мережевих затримок 45
3 АЛГОРИТМИ ОРГАНІЗАЦІЇ МЕРЕЖЕВОЇ ВЗАЄМОДІЇ В ІГРОВИХ ЗАСТОСУНКАХ 48
3.1 Опис проблеми у сучасних рішень.. 48
3.2 Структура серверного додатка 50
3.3 Використання бібліотек та фреймворків для мережевого програмування 54
3.4 Реалізація основних компонентів.. 56
3.5 Модель синхронізації об’єктів гри 67
3.6 Логування та діагностика мережевих подій.. 71
3.6.1 Призначення та необхідність логування 72
3.6.2 Реалізація системи логування 72
3.6.3 Діагностика та аналіз логів 73
3.6.4 Оптимізація та майбутнє розширення системи логування............... 73
4 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ТА ТЕСТУВАННЯ 74
4.1 Опис методології тестування мережевої взаємодії 74
4.2 Тестування продуктивності при різній кількості підключених клієнтів 75
4.3 Тестування продуктивності при різному мережевому навантаженні.. 80
4.4 Аналіз ефективності системи 84
ВИСНОВКИ... 86
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 88
ДОДАТОК А Графічний матеріал кваліфікаційної роботи............................ 91

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

CPU – Central Processing Unit

MMORPG – Massively Multiplayer Online Role-Playing Game

P2P – Peer-to-Peer

PvP – Player versus Player

RPC – Remote Procedure Calls

RTT – Round-Trip Time

TCP – Transmission Control Protocol

UDP – User Datagram Protocol

ВСТУП

Розробка мережевих компонентів для мультиплеєрних ігор є однією з ключевих задач у галузі ігрового програмування. Сучасні ігрові проєкти висувають все вищі вимоги до швидкості передачі даних, стабільності з'єднання та масштабованості мережевих рішень. Незважаючи на наявність стандартних інструментів у популярних ігрових рушіях, таких як Unity Multiplayer Netcode for GameObjects, існуючі рішення мають суттєві обмеження.

Аналіз практики розробки мультиплеєрних ігор свідчить, що стандартні мережеві рішення Unity задовольняють потреби лише простих проєктів з низькою та середньою інтенсивністю мережевої взаємодії. Для складних ігрових систем, особливо до жанрів де основою лежить часте оновлення даних вони демонструють недостатню пропускну здатність, або неефективне використання мережевих ресурсів.

На сьогодні у галузі мережевого програмування для Unity залишаються суттєві прогалини у питаннях оптимізованої серіалізації даних, ефективної обробки мережевих пакетів різних розмірів, та їх пріоритизація.

Відсутні також комплексні рішення, які б інтегрували передові підходи до оптимізації мережевого коду та були б достатньо гнучкими для адаптації під різні типи ігрових проєктів. Більшість доступних засобів або надто загальні, або вузькоспеціалізовані, що обмежує їх практичне застосування в широкому спектрі ігрових застосунків.

Метою кваліфікаційної роботи є розробка вдосконалених методів та алгоритмів організації мережевого програмування для мультиплеєрних ігрових систем на базі Unity, які дозволять подолати зазначені обмеження та забезпечать створення більш ефективних, гнучких та масштабованих мережевих рішень для сучасних ігрових проєктів.

Об'єктом дослідження є процеси мережевої взаємодії в мультиплеєрних

ігрових системах, розроблених на платформі Unity.

Предметом дослідження є методи та алгоритми оптимізації передачі даних у мережевому програмуванні ігрових застосунків.

Для досягнення поставленої мети необхідно розробити власні моделі та рішення для оптимізації мережевої взаємодії в ігрових системах. Потрібно створити вдосконалену систему серіалізації даних, яка дозволить зменшити обсяг мережевого трафіку без втрати інформативності. Розробка алгоритмів динамічного управління пакетами різного розміру забезпечить ефективне використання доступних мережевих ресурсів у різних умовах з'єднання. Впровадження системи пріоритизації мережевих повідомлень дозволить забезпечити своєчасну доставку критично важливих даних, тоді як менш термінова інформація може передаватися з нижчим пріоритетом.

1 СУЧАСНІ ТЕХНОЛОГІЇ ОРГАНІЗАЦІЇ МЕРЕЖЕВОГО ПРОГРАМУВАННЯ МУЛЬТИПЛЕЄРНИХ СИСТЕМ

1.1 Аналіз існуючих рішень для створення мультиплеєрної системи

Мультиплеєрні системи є основою сучасних ігрових застосунків, і майже невід’ємною частиною більшості ігрових проектів починаючи від повчальних ігор закінчуючи шутерами та стратегіями [1]. Вони дозволяють гравцям взаємодіяти в реальному часі. На сьогодні існує декілька популярних рішень для реалізації мережевого програмування, серед яких можна виділити можливість використання ігрових рушіїв із вбудованими мережевими можливостями, сторонні мережеві бібліотеки, а також самостійно розроблені серверні рішення. Так от наприклад ігровий рушій від компанії Unity надає такі можливості:

1.1.1 UNet (Unity Networking)

UNet (Unity Networking) – це застаріла мережева система, що набула значної популярності серед розробників мультиплеєрних ігор. Незважаючи на оголошення про припинення підтримки у 2018 році та поступову заміну на нову систему (Netcode for GameObjects), UNet все ще залишається важливим для аналізу як технологія, що вплинула на подальший розвиток мережевих рішень [16].

UNet базується на клієнт-серверній моделі з високорівневим API (HLAPI), що робить його відносно простим для впровадження навіть розробниками без глибоких знань мережевого програмування. Основу архітектури становлять NetworkManager для керування з'єднаннями та мережевими сесіями, NetworkIdentity для ідентифікації об'єктів у мережі та NetworkBehaviour для реалізації мережевої логіки в ігрових об'єктах.

Ключовими особливостями UNet є система синхронізації стану (State Synchronization) для автоматичної передачі змін між клієнтами, та віддалені виклики процедур (Remote Procedure Calls, RPC) для виконання дій на віддалених машинах. UNet також пропонує систему спавну мережевих об'єктів та механізм передачі влади над об'єктами між клієнтами.

Серед переваг UNet варто відзначити відносну простоту використання, інтеграцію з інспектором Unity, що дозволяє налаштовувати мережеві параметри без написання коду, та надійну роботу для невеликих проектів. Однак, система має суттєві обмеження у вигляді складнощів масштабування для великих проектів, обмеженої гнучкості при нестандартних сценаріях використання та високого навантаження на центральний сервер.

Після оголошення про припинення підтримки UNet, частина його функціональності та концепцій була перенесена в новіші рішення, наприклад в Netcode for GameObjects або Mirror, що дозволяє говорити про значний вплив UNet на подальший розвиток мережевих систем для Unity.

1.1.2 Photon

Photon – комерційна мережева платформа, яка забезпечує клієнт-серверну архітектуру та має можливості для масштабування. Вона складається з Photon Server, Photon Cloud та Photon Fusion, що надають інструменти для реалізації мультиплеєрних ігор з високою продуктивністю та низькими затримками [12]. Photon дозволяє використовувати як централізований сервер, так і розподілену P2P-архітектуру, підтримуючи механізми синхронізації станів, обробки запитів та управління підключеннями. Однією з ключових переваг є масштабованість у хмарі, що дає змогу легко адаптувати серверну інфраструктуру до потреб гри. Також не менш важливим є підтримка кросплатформеності, що дозволяє взаємодіяти користувачам з різних пристроїв. Можна додати, що розробник може повністю зосередитися на розробці гри, в той час як хостинг, робота сервера

та масштабування – всім цим опікується Photon Cloud. SaaS – одна з форм хмарних обчислень, модель обслуговування, за якої передплатникам надається готове прикладне програмне забезпечення, яке повністю обслуговує провайдер.

Процес для гравця є таким: підключення до лобі, де зберігаються номери кімнат на головному сервері. Photon unity network автоматично приєднується до цього лобі та надає гравцю можливість обрати кімнату з отриманого списку. Не обов'язково приєднуватись до лобі, можна відразу підключитися до випадкової кімнати або конкретної кімнати, якщо гравець знає її номер. Кімната – місце, де збираються гравці. Є можливість налаштувати її характеристики (наприклад, можна виставити обмеження кількості гравців, її номер та інше).

Підключення відбувається в одну стрічку за допомогою вбудованого статичного класу PhotonNetwork та скрипту PhotonServerSettings, який знаходиться за шляхом Assets\Photon Unity Networking\Resources. Цей клас має методи під'єднання до кімнат (Join Room) та їх створення (Create Room). Щоб додати об'єкт на сцену, потрібно мати на ньому компонент Photon View та використати метод PhotonNetwork.Instantiate. Однак може виникнути проблема, що всі інші об'єкти також будуть реагувати на керування. Щоб запобігти цьому потрібно вказати в якості батьківського класу об'єкта Photon.MonoBehaviour, та додати умову до скрипта керування, яка б запускала його тільки коли властивість photonView.isMine – істина.

Photon Cloud має велику кількість компонентів та методів для синхронізації. Наприклад за допомогою Photon Transform View дуже просто синхронізувати властивості об'єктів, які повинні передаватись по мережі під час гри. Щоб синхронізувати анімації достатньо додати Photon Animator View до об'єкта, та вказати необхідні шари та параметри аніматора. Photon Cloud пропонує дуже зручний спосіб роботи з віддаленими викликами. Загальним правилом є те, що RPC розуміє лише основні примітиви: int, float, bool, string тощо. Щоб увімкнути віддалений виклик для певного методу,

потрібно застосувати атрибут PunRPC [12]. Замість безпосереднього виклику цільового методу необхідно викликати RPC() на компоненті PhotonView і вказати ім'я методу, який потрібно викликати.

1.1.3 Mirror

Mirror – одне з найбільш поширених відкритих рішень для розробки мережевих ігор. Mirror виник як рішення спільноти UNET після того, як компанія Unity оголосила про припинення підтримки своєї вбудованої мережевої системи [10].

Mirror представляє собою високорівневий мережевий фреймворк, який забезпечує простий, але потужний API для розробки мультиплеєрних ігор. Основною перевагою Mirror є його проста інтеграція та орієнтованість на розробників, які не мають глибоких знань з мережевого програмування.

Архітектура Mirror базується на компонентному підході, де система складається з декількох ключових елементів. Центральним компонентом є NetworkManager, який відповідає за управління мережевими з'єднаннями, створення сесій та налаштування параметрів мережі [10]. NetworkIdentity є компонентом, що приєднується до ігрових об'єктів для їх ідентифікації в мережі. Для додавання мережевої функціональності до об'єктів використовується NetworkBehaviour, який є базовим класом для всіх скриптів, що потребують мережевої синхронізації.

Важливими особливостями Mirror є підтримка State Synchronization – механізму, що дозволяє автоматично синхронізувати стан об'єктів між сервером та клієнтами, та використання Commands і ClientRpcs для комунікації між клієнтами та сервером. Commands – це методи, які викликаються клієнтом та виконуються на сервері, тоді як ClientRpcs – методи, які викликаються сервером та виконуються на всіх клієнтах.

Mirror підтримує різні транспортні протоколи, включаючи TCP, UDP та WebSockets, що дозволяє використовувати його для розробки ігор на різних

платформах. Завдяки модульній архітектурі, розробники можуть легко замінювати транспортний шар без необхідності модифікації основного коду гри.

Система синхронізації в Mirror базується на концепції централізованого сервера, де він є джерелом достовірної інформації про стан гри. Це забезпечує захист від читерства (шахрайства) та дозволяє підтримувати консистентність ігрового світу на всіх клієнтах. Однак, це також означає, що клієнти мають обмежений контроль над ігровим процесом, що може призводити до затримок у відображенні результатів дій гравця.

Водночас, Mirror має і певні обмеження. Він може бути менш ефективним для великих мультиплеєрних ігор з високим рівнем навантаження. Крім того, система не надає вбудованих інструментів для масштабування серверів, що може бути проблемою для ігор з великою кількістю гравців.

У контексті застосування в індустрії, Mirror широко використовується для розробки інді-ігор, прототипів та середніх за розміром мультиплеєрних проектів. Він є популярним вибором для кооперативних ігор, ігор з відкритим світом невеликого масштабу та мультиплеєрних головоломок.

1.1.4 Netcode for GameObjects

Netcode for GameObjects – ця технологія є важливим компонентом Unity Transport Layer і представляє собою високорівневу систему для синхронізації ігрових об'єктів через мережу.

Розроблений для спрощення процесу створення мультиплеєрних ігор. Технологія базується на клієнт-серверній архітектурі, що забезпечує необхідний рівень надійності та безпеки мережевої взаємодії. Основними архітектурними компонентами даного рішення є NetworkManager, NetworkObject, NetworkBehaviour, NetworkVariable, для автоматичної синхронізації змінних між клієнтами та система RPC для виклику методів на

віддалених клієнтах.

Серед ключових переваг варто відзначити високий рівень абстракції, що дозволяє розробникам не занурюватися у низькорівневі деталі мережевої взаємодії. Тісна інтеграція з екосистемою Unity забезпечує можливість використання знайомих інструментів та робочих процесів. Автоматична синхронізація об'єктів між клієнтами суттєво спрощує розробку. Крім того, технологія забезпечує ефективне використання смуги пропускання завдяки вбудованим механізмам оптимізації мережевого трафіку.

Важливою особливістю є чітко визначена модель володіння об'єктами, де кожен об'єкт має власника, який контролює його стан. Це дозволяє уникнути конфліктів при одночасному доступі до об'єктів та забезпечує стабільність роботи системи.

Механізми синхронізації представлені двома основними підходами. Перший – використання NetworkVariable для автоматичної синхронізації значень між сервером та клієнтами. Другий – застосування RPC для виклику методів на віддалених клієнтах. Обидва підходи дозволяють гнучко налаштовувати систему під конкретні потреби гри.

Попри значні переваги, система має певні обмеження. Технологія найкраще працює з моделлю централізованого сервера, де він контролює стан гри. Для великих мультиплеєрних ігор з тисячами гравців може знадобитися додаткова оптимізація. Крім того, залежність від Unity обмежує використання цієї технології в інших середовищах розробки. При великій кількості об'єктів, що синхронізуються, можливе зростання мережевого трафіку, що потребує додаткової уваги з боку розробників.

Діапазон застосування технології починається від невеликих інді-ігор до середніх за розміром комерційних продуктів. Система особливо популярна для кооперативних ігор з невеликою кількістю гравців, PvP-ігор з обмеженою кількістю учасників на арені, а також мультиплеєрних головоломок та стратегій у реальному часі.

1.2 Класифікація мультиплеєрних архітектур

Мережева архітектура в контексті ігрових систем визначає структуру та принципи організації взаємодії між компонентами системи, які забезпечують функціонування мультиплеєрного режиму. Вона охоплює аспекти розподілу обчислювального навантаження, моделі комунікації, стратегії синхронізації стану гри та механізми забезпечення надійності передачі даних [8].

Історичний розвиток мультиплеєрних архітектур почався з простих локальних мереж, де ігри підтримували обмежену кількість гравців, і пройшов шлях до складних розподілених систем, здатних обслуговувати мільйони користувачів одночасно. З еволюцією мережевих технологій та зростанням обчислювальних потужностей, архітектури мультиплеєрних систем ставали все більш складними та диверсифікованими, відповідаючи на нові виклики та можливості.

Класифікація мережевих архітектур може здійснюватися за різними критеріями, серед яких найбільш значущими є:

- модель розподілу авторитету (визначення джерела істини у грі);
- топологія мережевих з'єднань;
- масштабованість;
- стійкість до затримок;
- безпека та захист від шахрайства;
- складність реалізації та підтримки.

1.2.1 Клієнт-серверна архітектура

Клієнт-серверна архітектура є одним з найпоширеніших підходів до організації мультиплеєрної взаємодії [2, 9]. В її основі лежить концепція централізованого управління ігровим процесом через виділений сервер, який

виступає як єдине джерело істини щодо стану гри (рисунок 1.1).

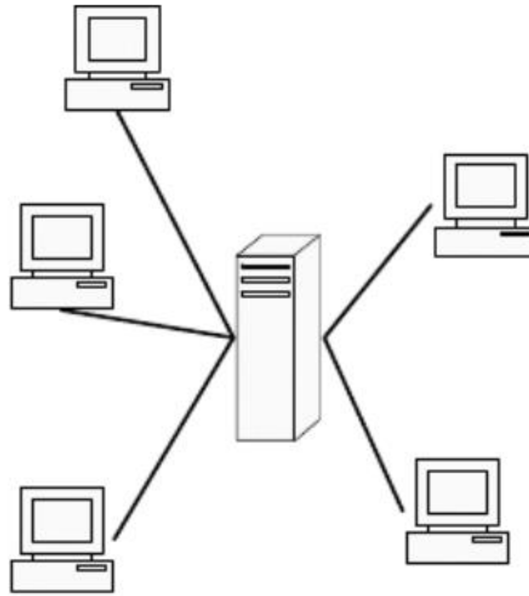

Рисунок 1.1 – Клієнт-серверна архітектура

Принцип роботи полягає в чіткому розподілі функцій між сервером та клієнтськими програмами. Сервер відповідає за обробку логіки гри, підтримку цілісного стану ігрового світу, координацію дій гравців та валідацію їхніх дій. Клієнти, у свою чергу, відповідають за відображення ігрового світу користувачеві, обробку введення та передачу команд на сервер. Така архітектура реалізує модель авторитарного сервера, де всі рішення щодо зміни стану гри приймаються централізовано [9].

Авторитарний сервер має вирішальне значення для забезпечення чесності гри, оскільки всі дії гравців проходять перевірку на сервері перед тим, як вплинути на загальний стан гри. Це суттєво ускладнює шахрайство та несанкціоновану модифікацію ігрового процесу.

Якщо говорити про переваги клієнт-серверної архітектури, то, по-перше, тут наявний високий рівень контролю над ігровим процесом. По-друге, є спрощення протидії шахрайству та простіша модель безпеки. Через можливість централізованого зберігання даних, існує єдина точка

синхронізації стану гри [15].

Проте, попри такі чудові характеристики, можна виділити і негативні сторони. По-перше, висока вартість інфраструктури для масштабних проектів. По-друге, вразливість до відмови центрального вузла, що може призвести до затримки при комунікації між клієнтом і сервером. По-третє, необхідність постійного підключення до мережі.

Також, з часом, така архітектура набула декілька варіацій: з виділеним сервером та з клієнт–хостом. Перша, функціонує на спеціалізованому обладнанні та управляється розробником або видавцем гри. Друга ж, працює у режимі, де один із клієнтів тимчасово бере на себе функції сервера.

Прикладами ігор, що використовують клієнт-серверну архітектуру, є World of Warcraft, Counter-Strike, Fortnite, League of Legends та більшість сучасних масових онлайн-ігор.

1.2.2 Peer-to-peer архітектура

На противагу централізованому підходу, архітектура peer-to-peer (P2P) передбачає пряму взаємодію між клієнтами без необхідності центрального сервера. У такій архітектурі кожен учасник мережі є одночасно і клієнтом, і сервером, що забезпечує розподілений характер системи (рисунок 1.2).

Принцип роботи P2P-архітектури базується на встановленні прямих з'єднань між вузлами мережі та обміні даними без посередництва. Кожен вузол відповідає за обробку частини загальної логіки гри та синхронізацію свого стану з іншими учасниками [9, 15].

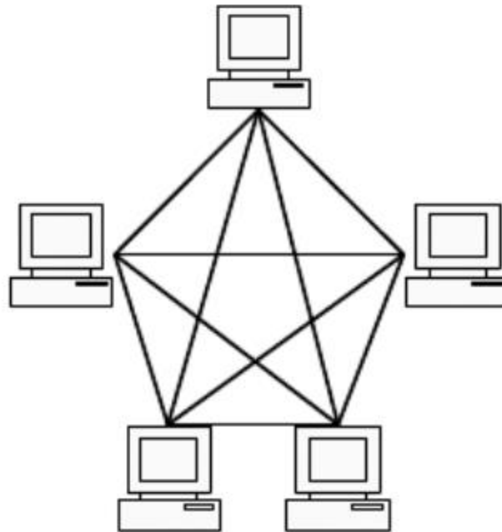

Рисунок 1.2 – Peer-to-peer архітектура

Розподілений авторитет є ключовою особливістю P2P-систем. У найпростішому випадку, кожен клієнт має повний авторитет над своїми об'єктами та діями, а інші клієнти приймають ці дані як достовірні. У більш складних реалізаціях може застосовуватися узгоджений підхід, при якому зміни стану гри приймаються лише після узгодження між декількома або всіма учасниками [9, 13].

У такої архітектури також є певні плюси та мінуси у використанні. По-перше, відсутність потреби у виділеній серверній інфраструктурі, через що випливає потенційно менші затримки при прямій комунікації. По–друге, висока стійкість до відмови окремих вузлів. Також з плюсів, можна визначити можливість функціонування в умовах обмеженого доступу до глобальної мережі [13,15].

Говорячи про негативні сторони, то тут здебільшого виділяють складність забезпечення синхронізації між усіма вузлами, через що система стає більш вразливою до шахрайства. По-друге, складність масштабування на велику кількість гравців та залежність від якості з'єднання між окремими вузлами [17].

З часом, через певні нові потреби, виникли нові підвиди цієї архітектури: чистий та гібридний. Де перший базується на тому, що всі вузли

мають рівнозначні ролі та повністю рівноправні. А другий – де деякі функції (наприклад, початкове з'єднання гравців або валідація критичних дій) делегуються центральному компоненту [17].

Приклади ігор з P2P-архітектурою включають ранні версії Doom та Quake для мультиплеєрного режиму, Age of Empires, а також деякі консольні ігри, такі як ранні серії Call of Duty на консолях [13].

1.2.3 Гібридні архітектури

З розвитком мережевих технологій та ускладненням вимог до ігрових систем, все більшої популярності набувають гібридні архітектури, які комбінують елементи клієнт-серверного та P2P підходів для отримання оптимального балансу між їхніми перевагами (рисунок 1.3) [14, 15].

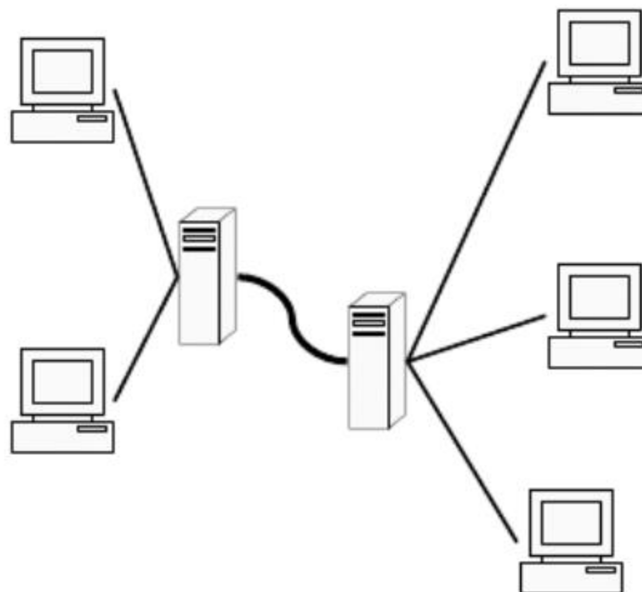

Рисунок 1.3 – Приклад гібридної архітектури

Комбінування клієнт-серверного та P2P підходів може здійснюватися різними способами. Найчастіше центральний сервер використовується для авторитетної валідації критичних дій (таких як нарахування очок або економічні транзакції), тоді як некритичні взаємодії між гравцями, які

потребують мінімальної затримки (наприклад, позиціонування персонажів), здійснюються через прямі P2P-з'єднання.

Особливості реалізації пов'язані з необхідністю чіткого розмежування відповідальності між централізованими та розподіленими компонентами системи. Це вимагає розробки складних протоколів синхронізації та механізмів узгодження даних, отриманих з різних джерел.

Особливо використання такого підходу є оптимальним, коли необхідно забезпечити мінімальні затримки при взаємодії між гравцями. Також іншим аспектом є потреба у збереженні централізованого контролю над критичними частинами гри та обмеження на масштабування серверної інфраструктури. Іншою причиною може стати важливість забезпечити стійкість системи до тимчасових проблем із з'єднанням.

Прикладами ігор, що використовують гібридні архітектури, є деякі файтинги, гоночні симулятори та шутери, де позиціонування та швидка взаємодія можуть здійснюватися через P2P, а облік досягнень та прогресу гравців – через центральний сервер.

1.2.4 Архітектура з розподіленим авторитетом

Архітектура з розподіленим авторитетом є особливим підходом до організації мультиплеєрної взаємодії, при якому права на визначення стану різних компонентів гри делегуються різним учасникам системи.

Принципи делегування авторитету в таких системах базуються на розподілі відповідальності відповідно до специфіки ігрових механік та вимог до швидкодії. Найчастіше використовується підхід, при якому кожен клієнт має повний авторитет над своїми персонажами або об'єктами, а інші клієнти приймають ці дані як достовірні. Це дозволяє мінімізувати затримки при взаємодії з керованими гравцем об'єктами [15].

Зональний підхід до розподілу авторитету передбачає розбиття ігрового світу на окремі зони, кожна з яких знаходиться під контролем

певного вузла мережі. Такий підхід дозволяє ефективно розподілити обчислювальне навантаження та забезпечити масштабованість системи за рахунок локалізації взаємодій.

Особливості синхронізації в архітектурах з розподіленим авторитетом пов'язані з необхідністю узгодження потенційно конфліктуючих змін стану. Для цього використовуються спеціальні алгоритми узгодження, часових міток та розв'язання конфліктів, які дозволяють підтримувати узгоджений стан гри навіть у умовах затримок передачі даних та нестабільного з'єднання.

1.2.5 Архітектури для масштабних онлайн-ігор

Особливим класом мультиплеєрних архітектур є рішення, орієнтовані на підтримку масштабних онлайн-ігор з тисячами або мільйонами одночасних користувачів. Такі архітектури характеризуються наявністю складних механізмів масштабування та розподілу навантаження [17].

Шардинг є одним з ключових підходів до масштабування і передбачає розділення ігрового світу на незалежні "шарди" або сервери, кожен з яких обслуговує окрему групу гравців (рисунок 1.4). Шарди можуть бути повністю ізольованими або дозволяти обмежену взаємодію між собою. Прикладами ігор, що використовують шардинг, є World of Warcraft та EVE Online.

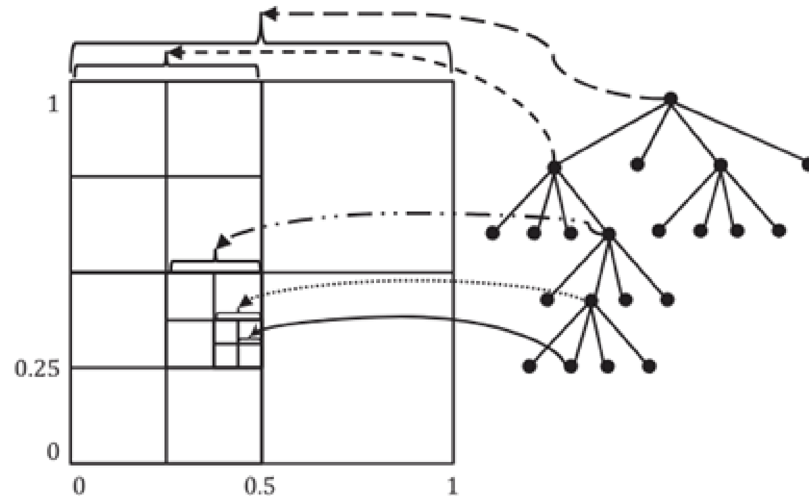

Рисунок 1.4 – Графічний вигляд ігрового світу, що поділено на вкладені регіони, які представлено деревом

Інстансинг представляє собою підхід, при якому створюються тимчасові копії певних зон ігрового світу для окремих гравців або груп. Це дозволяє забезпечити оптимальне навантаження на сервери та уникнути переповнення ігрових локацій. Даний підхід широко використовується в MMORPG для організації підземель, рейдів та інших групових активностей [3].

Балансування навантаження є критичним аспектом архітектур для масштабних онлайн-ігор і реалізується через динамічний розподіл гравців між серверами, адаптивне виділення ресурсів та перерозподіл обчислювального навантаження. Сучасні системи балансування можуть враховувати географічне розташування гравців, поточне навантаження на сервери та характер ігрових активностей [3, 17].

Реплікація даних забезпечує надійність та доступність ігрових даних через утворення множинних копій на різних серверах. У контексті масштабних онлайн-ігор особливо важливими є механізми синхронізації реплік, які повинні забезпечувати узгодженість даних при мінімальних затримках.

1.2.6 Критерії вибору архітектури для конкретного проекту

Вибір оптимальної мультиплеєрної архітектури для конкретного проекту є комплексним завданням, яке вимагає врахування множини факторів та компромісів [3, 9].

Вимоги до затримки є одним з ключових критеріїв, який визначає, наскільки критичною для ігрового процесу є швидкість взаємодії між гравцями. Для динамічних жанрів, таких як шутери або файтинги, прийнятними вважаються затримки не більше 50-100 мс, тоді як для покрокових стратегій допустимі затримки можуть складати декілька секунд.

Масштабованість визначає здатність архітектури ефективно функціонувати при збільшенні кількості одночасних гравців. Для проектів з очікуваною великою аудиторією критичними є механізми горизонтального масштабування та балансування навантаження.

Безпека є невід'ємним аспектом будь-якої мультиплеєрної архітектури і включає захист від шахрайства, несанкціонованого доступу та маніпуляцій із даними. Різні архітектури пропонують різні моделі безпеки, від авторитарного контролю у клієнт-серверних системах до розподілених механізмів валідації у P2P-мережах.

Складність реалізації відображає технічні виклики, пов'язані з розробкою та підтримкою обраної архітектури. Вона залежить від наявності готових рішень та інструментів, досвіду команди та специфічних вимог проекту.

Економічні фактори часто є визначальними при виборі архітектури, особливо для незалежних розробників та стартапів. Вони включають вартість розробки, витрати на підтримку інфраструктури та масштабування, а також можливості монетизації, пов'язані з обраною архітектурою.

Комплексний підхід до вибору архітектури передбачає аналіз усіх зазначених критеріїв у контексті конкретних вимог проекту та доступних ресурсів. У багатьох випадках оптимальним є гібридне рішення, яке комбінує

елементи різних архітектур для досягнення найкращого балансу між продуктивністю, надійністю, безпекою та вартістю.

1.3 Протоколи передачі даних у мультиплеєрних системах

Ефективність функціонування мультиплеєрних систем значною мірою залежить від методів і технологій передачі даних між компонентами гри. Протоколи передачі даних відіграють ключову роль у забезпеченні стабільної, швидкої та захищеної комунікації між клієнтами та серверами, що безпосередньо впливає на якість користувацького досвіду [3].

Варто зазначити, що вибір відповідних протоколів передачі даних є критичним фактором для успішної реалізації мультиплеєрної системи та безпосередньо впливає на такі характеристики як затримка, пропускна здатність, масштабованість та стабільність ігрового процесу. У контексті сучасних мультиплеєрних ігор особливої уваги вимагають питання мінімізації затримок, ефективного використання мережевих ресурсів та забезпечення стійкості до різноманітних мережевих умов.

1.3.1 Основи мережевої взаємодії в мультиплеєрних системах

Протоколи передачі даних у мультиплеєрних системах, забезпечують обмін інформацією між географічно розподіленими учасниками гри. Вони слугують абстрактним шаром між програмним забезпеченням гри та мережевою інфраструктурою, інкапсулюючи складність мережевої взаємодії та надаючи розробникам стандартизований інтерфейс для обміну даними.

Значення протоколів у мультиплеєрних системах виходить далеко за межі простої передачі інформації. Вони визначають такі фундаментальні аспекти як структуру комунікації, формат даних, механізми забезпечення надійності та методи вирішення проблем, пов'язаних із мережевою нестабільністю. В умовах конкурентних мультиплеєрних ігор, де

синхронізація ігрових дій між клієнтами має відбуватися в реальному часі, протоколи повинні забезпечувати мінімальну затримку при збереженні узгодженості ігрового стану.

Ефективність протоколів передачі даних у мультиплеєрних іграх оцінюється за низкою критеріїв, серед яких затримка – час, необхідний для передачі пакета даних від відправника до отримувача. У динамічних іграх допустимою вважається затримка до 100 мс, що забезпечує відчуття реальної взаємодії. Не менш важливими є пропускна здатність – обсяг даних, що може бути переданий за одиницю часу, що особливо важливо для масових онлайн-ігор з великою кількістю одночасних з'єднань. Та надійність – здатність протоколу забезпечувати доставку всіх пакетів даних без втрат, що критично для передачі важливих ігрових подій. Окремої уваги заслуговують масштабованість – здатність ефективно працювати при збільшенні кількості клієнтів або обсягу даних. Стійкість до мережевих проблем – можливість зберігати функціональність в умовах нестабільного з'єднання, втрати пакетів або коливань затримки. Та ефективність використання ресурсів – мінімізація накладних витрат на службові дані, що особливо важливо для мобільних пристроїв з обмеженим трафіком.

Вибір протоколу суттєво впливає на користувацький досвід у мультиплеєрних іграх. Неоптимально обраний протокол може призвести до таких негативних явищ як "лаги" (затримки у відображенні дій інших гравців), "фризи" (короткочасні зупинки гри), десинхронізація ігрового стану між клієнтами та відчуття затримки керування. У іграх, що вимагають точності та швидкої реакції, навіть незначні затримки можуть критично позначитися на ігровому процесі, тоді як у покрокових стратегіях вимоги до мінімізації затримки можуть бути менш критичними.

Транспортні протоколи забезпечують базовий механізм передачі даних між вузлами мережі та формують основу для функціонування мультиплеєрної взаємодії. Найпоширенішими транспортними протоколами в контексті мультиплеєрних систем є TCP та UDP, кожен з яких має

специфічні характеристики, що визначають сфери їхнього оптимального застосування [3, 11].

1.3.2 TCP (Transmission Control Protocol)

TCP є одним із фундаментальних протоколів стеку TCP/IP та характеризується орієнтованістю на з'єднання, що забезпечує надійну передачу даних. Функціонування TCP базується на механізмі "трикрокового рукостискання" (рисунок 1.5) для встановлення з'єднання та підтвердження отримання пакетів, що гарантує цілісність даних [9, 11].

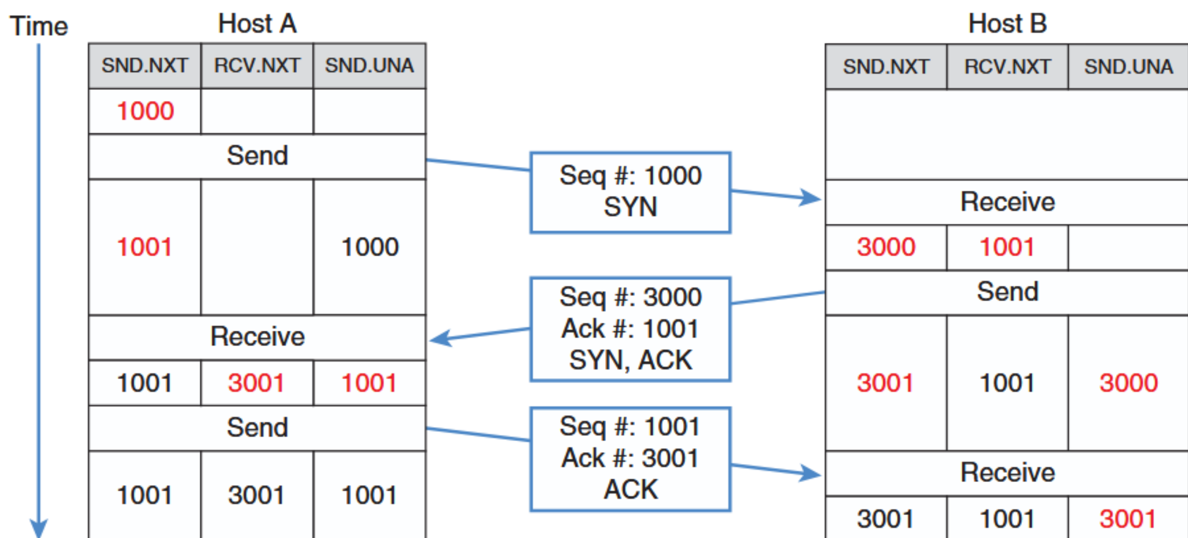

Рисунок 1.5 – Трикрокове рукостискання TCP

Основними перевагами TCP для мультиплеєрних систем є гарантована доставка даних – протокол забезпечує повторну передачу втрачених пакетів, що критично для передачі важливих ігрових подій та оновлень стану. Збереження порядку пакетів – TCP гарантує, що пакети будуть оброблені отримувачем у тому ж порядку, в якому вони були відправлені, що спрощує логіку синхронізації стану гри. Контроль перевантаження – адаптивне регулювання швидкості передачі для запобігання перевантаження мережі. А

також простота використання – більшість мережевих бібліотек та фреймворків надають зручний інтерфейс для роботи з TCP.

Незважаючи на переваги, TCP має суттєві обмеження для деяких типів мультиплеєрних ігор, наприклад підвищену затримку – механізми трикрокового рукостискання та повторної передачі втрачених пакетів збільшують час доставки даних. Проблему Head-of-Line Blocking – затримка або втрата одного пакета блокує обробку всіх наступних пакетів до вирішення проблеми, що може призводити до "фризів" у грі. Значні накладні витрати – службова інформація в заголовках пакетів та механізми підтвердження збільшують обсяг трафіку (рисунок 1.6). А також проблеми з NAT-проходженням – складність встановлення прямих з'єднань між клієнтами за NAT, що важливо для архітектури клієнт-клієнт.

Bits	0	4	7	16
0–31	Source Port			Destination Port
32–63	Sequence Number			
64–95	Acknowledgment Number			
96–127	Data Offset	Reserved	Control Bits	Receive Window
128–159	Checksum			Urgent Pointer
160–...	Options			

Рисунок 1.6 – Заголовок TCP

У мультиплеєрних іграх TCP найчастіше використовується в сценаріях, де надійність передачі даних має пріоритет над мінімальною затримкою. Це включає стратегії реального часу з великою кількістю одиниць та складною логікою стану гри; масові онлайн-ігри з інтенсивним обміном даними між сервером і клієнтами; сценарії авторизації, початкової синхронізації стану та передачі критичних ігрових подій; а також браузерні мультиплеєрні ігри, що використовують WebSocket на основі TCP.

Яскравими прикладами ігор, що використовують TCP як основний транспортний протокол, є World of Warcraft, Eve Online та інші масові

багатокористувацькі онлайн-ігри, де узгодженість стану гри має пріоритет над мінімізацією затримок.

1.3.3 UDP (User Datagram Protocol)

UDP є альтернативним підходом до передачі даних, сфокусований на мінімізації затримок за рахунок відмови від механізмів забезпечення надійності. На відміну від TCP, UDP є протоколом без встановлення з'єднання та працює за принципом "відправив і забув", що дозволяє суттєво знизити затримку передачі [9, 11].

Ключовими перевагами UDP в контексті мультиплеєрних ігор є мінімальна затримка – відсутність механізмів встановлення з'єднання, підтвердження доставки та контролю перевантаження дозволяє досягти мінімального часу передачі даних. Також низькі накладні витрати – компактні заголовки пакетів зменшують обсяг службової інформації (рисунок 1.7). Відсутність блокування – втрата одного пакета не впливає на обробку інших, що забезпечує безперервність ігрового процесу. Не менш важливим є підтримка широкомовної та багатоадресної розсилки – можливість ефективної передачі даних одночасно багатьом клієнтам. Також є краща ефективність у нестабільних мережах – відсутність затримок на повторну передачу в умовах високих втрат пакетів.

Bits	0	16
0–31	Source Port	Destination Port
32–63	Length	Checksum

Рисунок 1.7 – Заголовок UDP

Однак використання UDP супроводжується рядом суттєвих недоліків, серед яких відсутність гарантій доставки – пакети можуть бути втрачені без будь-якого повідомлення, відсутність збереження порядку – пакети можуть

приходити в довільному порядку, відсутність контролю перевантаження – необхідність самостійної реалізації механізмів регулювання інтенсивності передачі. Також можна виділити складність розробки – необхідність реалізації власних механізмів підтвердження, обробки втрат та забезпечення порядку пакетів.

UDP є пріоритетним вибором для ігор, де мінімальна затримка критично важлива для якісного ігрового досвіду. Прикладами таких ігор є шутери від першої особи (Valorant, Counter-Strike, Apex Legends), файтинги та спортивні симулятори (FIFA, NFL), гоночні симулятори (Forza, Need for Speed), а також ігри з великою кількістю одночасних користувачів та високою частотою оновлень (Fortnite, PUBG). Але такий вибір часто може спровокувати виникнення таких ефектів як rubberbanding.

1.3.4 Гібридні рішення на основі TCP/UDP

Зважаючи на обмеження чистих TCP та UDP, у сучасних мультиплеєрних системах часто застосовуються гібридні підходи, що комбінують переваги обох протоколів. Такі рішення дозволяють диференціювати трафік залежно від вимог до надійності та затримки [17].

Основні підходи до реалізації гібридних рішень включають паралельне використання обох протоколів – критичні дані (авторизація, транзакції, важливі ігрові події) передаються через TCP, а нечутливі дані (позиції гравців, стан руху, ввід користувача) через UDP. Також реалізацію надійного шару над UDP – створення власних механізмів підтвердження доставки та контролю порядку для критичних пакетів при загальному використанні UDP, як транспортного протоколу. Іншим прикладом є застосування спеціалізованих бібліотек – використання готових рішень, таких як ENet, KCP або RakNet, що реалізують гнучкі механізми надійності на базі UDP. Також необхідно пам’ятати про адаптивну маршрутизацію – динамічний вибір протоколу залежно від типу даних, мережевих умов та вимог до

затримки.

Гібридні рішення широко застосовуються в сучасних комерційних іграх. Наприклад, Overwatch використовує TCP для передачі стану гри та UDP для передачі позицій та дій гравців. Minecraft застосовує власний протокол на базі TCP для передачі стану світу та змін у ньому, але реалізує оптимізації для зменшення впливу затримок.

Важливо відзначити, що реалізація гібридних рішень підвищує складність системи та вимагає ретельного проектування та тестування для забезпечення коректної взаємодії між різними каналами передачі даних.

1.4 Постановка завдань дослідження

У процесі створення мультиплеєрного ігрового застосунку на рушії Unity часто використовуються стандартні інструменти для організації мережевої взаємодії, в Netcode for GameObjects. Проте в більшості випадків цей фреймворк не забезпечує необхідного рівня гнучкості та швидкодії для проєктів, які потребують частого обміну даними між клієнтами або мають велику кількість одночасних гравців. На практиці це призводить до перевантаження мережі, появи затримок, втрати пакетів та неузгодженості стану ігрових об’єктів.

Більшість розробників на ранніх етапах зосереджуються на реалізації функціональності, відкладаючи оптимізацію мережевої частини на пізніше. Як наслідок – зі зростанням складності гри з'являються проблеми зі стабільністю з’єднання, ефективністю обміну повідомленнями та загальною продуктивністю мережевого модуля. Враховуючи, що мережевий обмін є критично важливою частиною будь-якого мультиплеєрного застосунку, така стратегія значно ускладнює масштабування системи та погіршує ігровий досвід користувача.

Мета цього дослідження полягає у створенні більш ефективної моделі організації мережевої передачі даних, що дозволить зменшити затримки,

знизити обсяг переданих даних та забезпечити плавну синхронізацію станів між усіма учасниками гри. Розробка охоплює побудову кастомізованої структури передачі повідомлень, альтернативних механізмів серіалізації та оптимізованих алгоритмів реплікації.

Основними проблемами, що спонукали до дослідження та створення вдосконаленого підходу, стали:

- повільна та надлишкова передача даних у стандартному Netcode for GameObjects;
- надмірне навантаження на CPU через часту обробку мережевих повідомлень;
- відсутність механізмів тонкого контролю над синхронізацією об’єктів;
- потреба в адаптивності мережевого ядра під специфіку гри (наприклад, високу динаміку, часті колізії чи велику кількість гравців);
- складність в інтеграції Netcode у нестандартні архітектури проєкту.

У зв’язку з цим виникла необхідність побудови власного рішення для організації мережевої взаємодії, яке буде орієнтоване на ефективність, масштабованість та мінімальні витрати ресурсів при високому навантаженні.

2 МЕТОДИ ОРГАНІЗАЦІЇ МЕРЕЖЕВОЇ ВЗАЄМОДІЇ В ІГРОВИХ ЗАСТОСУНКАХ

2.1 Архітектура мережевого коду

На сьогодні у контексті розробки мультиплеєрних ігрових застосунків архітектура мережевого коду відіграє важливу роль та визначає ефективність функціонування всієї системи. Коректно спроектована мережева архітектура забезпечує основу для всіх інших компонентів мультиплеєрного ігрового середовища, формуючи ядро для подальшої імплементації методів синхронізації, обробки та передачі даних.

Архітектура мережевого коду є комплексною системою взаємопов'язаних компонентів, що забезпечують комунікацію між різними ігровими клієнтами та серверами в режимі реального часу. Сучасні підходи до організації мережевої архітектури базуються на багаторівневій структурі абстракцій, що дозволяє ефективно керувати потоками даних та ізолювати функціональні аспекти мережевої взаємодії [9].

Основною складовою архітектури мережевого коду є транспортний рівень, рівень сесій та рівень прикладної логіки. Транспортний рівень відповідає за безпосередню передачу даних між вузлами мережі, працюючи з протоколами TCP/UDP [9, 11]. Рівень сесій забезпечує абстракцію над транспортним рівнем, інкапсулюючи логіку керування мережевими сесіями, аутентифікацію користувачів та підтримку стану з'єднання. Рівень прикладної логіки інтегрує мережеву архітектуру з ігровими механіками, визначаючи, які дані потребують синхронізації між клієнтами та яким чином це має відбуватися.

При проектуванні архітектури мережевого коду особливої уваги потребує встановлення балансу між різними архітектурними парадигмами, серед яких найбільш поширеними є клієнт-серверна та пірингова (peer-to-

peer) архітектури.

Важлива деталь архітектури мережевого коду є модель реплікації стану гри (рисунок 2.1). Виділяють два основних підходи: детерміністична реплікація та реплікація стану [3]. Детерміністична реплікація базується на передачі тільки користувацьких дій та їх послідовному відтворенні на всіх клієнтах, що значно зменшує обсяг мережевого трафіку, але вимагає повної детермінованості ігрової логіки. Реплікація стану передбачає безпосередню передачу змін ігрового стану, що спрощує розробку та зменшує вимоги до детермінованості, проте створює більше навантаження на мережу.

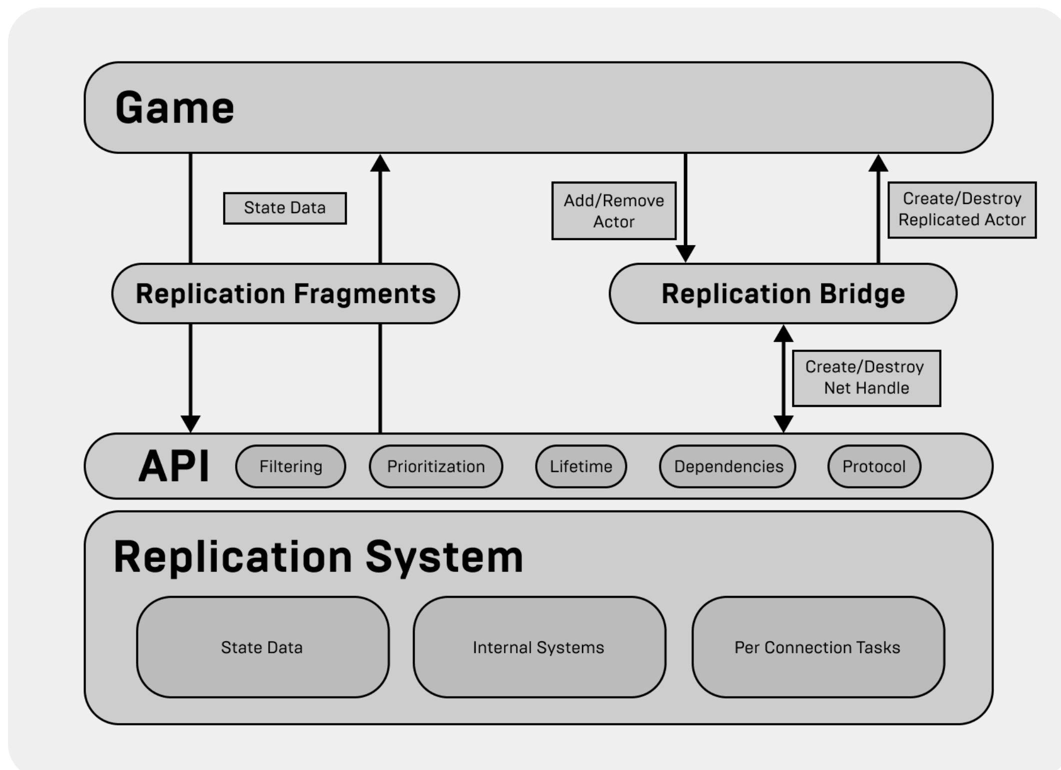

Рисунок 2.1 – Модель реплікації стану гри

Архітектурні рішення щодо серіалізації та десеріалізації даних також мають критичне значення для ефективності мережевого коду. Вибір між текстовими форматами (JSON, XML) та бінарними протоколами (Protocol Buffers, FlatBuffers) впливає на швидкість обробки даних, їх розмір та гнучкість модифікації структур. Бінарні протоколи забезпечують компактність та ефективність, що особливо важливо для мультиплеєрних

ігор, де мінімізація розміру пакетів має прямий вплив на продуктивність (рисунок 2.2).

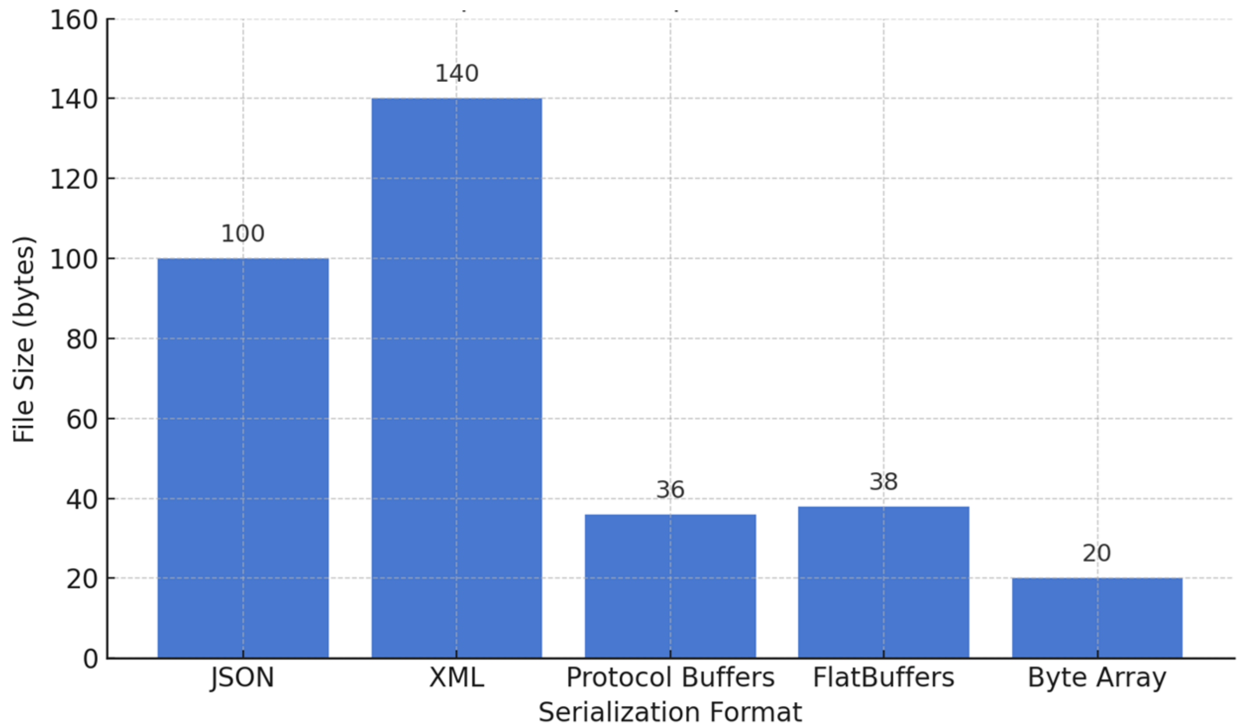

Рисунок 2.2 – Порівняння розмірів для різних форматів серіалізації

Інтеграція мережевого коду з ігровим рушієм становить окремий архітектурний виклик. В ідеальній архітектурі мережевий код повинен бути відокремленим від логіки ігрового рушія, забезпечуючи можливість повторного використання компонентів. Патерни проектування, такі як "Посередник" (Mediator), "Спостерігач" (Observer) та "Команда" (Command), часто використовуються для створення гнучких інтерфейсів між мережевим кодом та іншими підсистемами ігрового застосунку [7].

Особливої уваги в архітектурі мережевого коду потребує обробка асинхронності та конкурентності. Реалізація асинхронних операцій введення-виведення (async I/O) та ефективне управління потоками дозволяють мінімізувати блокування головного потоку гри та забезпечити плавність ігрового процесу. Сучасні підходи включають використання неблокуючих API, обробників подій та пулів потоків для оптимізації мережевих операцій.

Архітектура мережевого коду також має враховувати особливості різних апаратних платформ та мережевих середовищ. Кросплатформена сумісність вимагає абстрагування від специфічних особливостей операційних систем та мережевих стеків, а адаптивність до різноманітних умов мережі (високі затримки, втрати пакетів, обмежена пропускна здатність) є необхідною для забезпечення стабільного користувацького досвіду.

Модульність та розширюваність архітектури мережевого коду дозволяють ефективно інтегрувати нові функціональні вимоги та адаптуватися до змін у проекті. Застосування патернів ін'єкції залежностей та використання інтерфейсів забезпечують гнучкість архітектури та спрощують процес тестування окремих компонентів.

2.2 Методи забезпечення синхронізації та відмовостійкості

Забезпечення ефективної синхронізації та відмовостійкості становить фундаментальне завдання у розробці мережевих компонентів мультиплеєрних ігрових застосунків. Реалізація стабільного ігрового процесу в умовах непередбачуваного мережевого середовища вимагає впровадження комплексу методів та алгоритмів, що забезпечують постійність ігрового стану та стійкість до збоїв.

Синхронізація стану гри між усіма учасниками мультиплеєрної сесії є критичним випробовуванням, від якого напряму залежить консистентність ігрового досвіду. У сучасній практиці розробки виділяють ряд підходів до вирішення проблеми синхронізації, кожен з яких має власні переваги та обмеження.

Синхронізація на основі блокування (Lockstep Synchronization) представляє собою класичний підхід, при якому гра просувається дискретними кроками або фреймами, і кожен наступний крок починається лише після отримання підтвердження готовності від усіх клієнтів (рисунок 2.3) [17]. Даний метод забезпечує повну синхронізацію стану, що особливо

важливо для стратегічних ігор та симуляцій, де точність взаємодії елементів критично важлива. Недоліком такого підходу є висока чутливість до затримок у мережі – один клієнт з повільним з'єднанням може спричинити затримки для всіх учасників. Для мінімізації цього недоліку застосовуються техніки буферизації вхідних даних та прогнозування наступних станів.

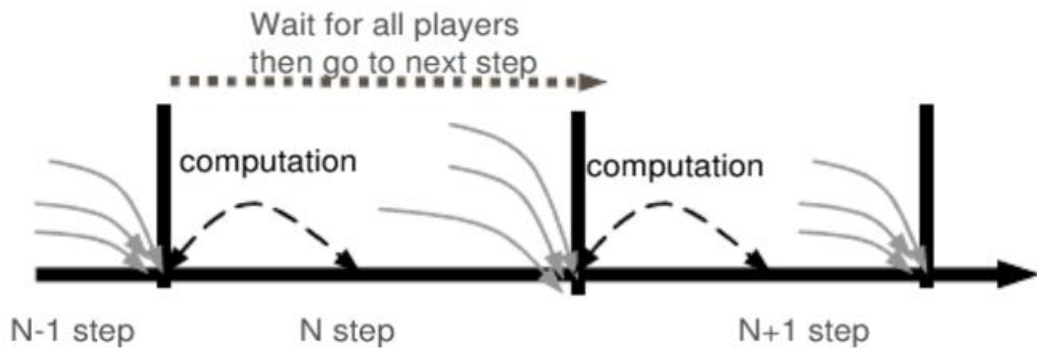

Рисунок 2.3 – Синхронізація на основі блокування

Авторитарна синхронізація (Authoritative Server Synchronization) базується на централізованій моделі, де сервер виступає єдиним джерелом істини щодо стану гри [9]. Клієнти надсилають дії користувачів на сервер, який виконує всі обчислення ігрової логіки та надсилає актуалізований стан назад до клієнтів. Цей підхід забезпечує високий рівень захисту від шахрайства та спрощує процес синхронізації, оскільки конфлікти вирішуються центральним авторитетом. Основним викликом при використанні авторитарної синхронізації є мінімізація відчуття затримки для користувача, особливо в іграх, що вимагають швидкої реакції. Для подолання цього обмеження широко застосовуються методи клієнтського передбачення (Client-Side Prediction) та згладжування (Smoothing).

Клієнтське передбачення дозволяє локальному клієнту тимчасово застосовувати дії користувача до локального стану, не чекаючи підтвердження від сервера [5]. Після отримання авторитетного стану від сервера клієнт здійснює узгодження (Reconciliation), коригуючи локальний

стан при необхідності. Цей метод значно покращує чуйність інтерфейсу, проте вимагає ретельної реалізації процедур узгодження для мінімізації візуальних артефактів при корекції стану.

У контексті синхронізації часу особливої уваги заслуговують методи компенсації затримок (Lag Compensation), що дозволяють враховувати мережеві затримки при обробці дій користувачів [11, 14]. Сервер може "повертатися в минуле", реконструюючи стан гри на момент, коли користувач фактично виконав дію, що значно підвищує точність взаємодії, особливо в динамічних іграх з швидким темпом (рисунок 2.4).

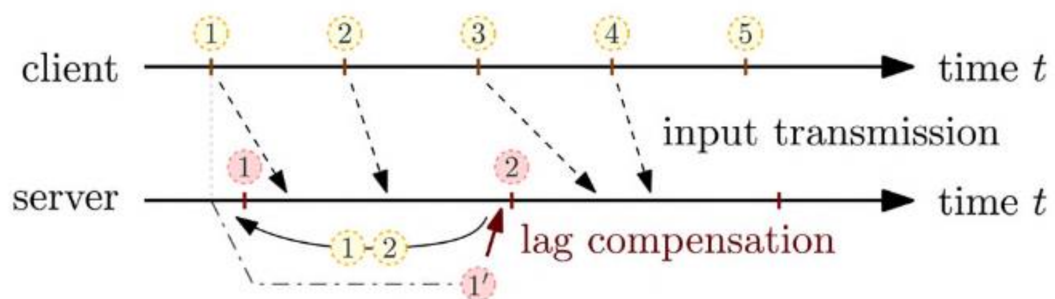

Рисунок 2.4 – Метод компенсації затримок

Часовий штамп (Time Stamping) є методом, при якому кожна подія маркується часовою міткою, що дозволяє відтворювати послідовність подій у правильному порядку, незалежно від порядку їх отримання. Цей підхід є основою для реалізації систем повторів (Replay Systems) та архівування ігрових сесій [9].

Відмовостійкість у мережевих ігрових застосунках охоплює широкий спектр методів, спрямованих на забезпечення безперервного функціонування системи навіть в умовах нестабільних з'єднань та апаратних відмов. Фундаментальним аспектом відмовостійкості є механізми виявлення та обробки розривів з'єднання.

Механізми виявлення розривів з'єднання базуються на систематичному обміні службовими повідомленнями (heartbeat messages) між клієнтами та сервером. Відсутність очікуваних повідомлень протягом визначеного

часового інтервалу сигналізує про потенційний розрив з'єднання та ініціює процедури відновлення або компенсації.

Техніки збереження стану (State Preservation) забезпечують можливість відновлення гри після тимчасових збоїв у з'єднанні. Періодичне збереження контрольних точок (Checkpoints) стану гри дозволяє швидко відновити сесію у випадку переривання, а інкрементальне збереження змін між контрольними точками мінімізує обсяг даних, що підлягають синхронізації при відновленні з'єднання.

Реплікація серверів є потужним методом забезпечення відмовостійкості на рівні інфраструктури. Активно-пасивна реплікація передбачає наявність резервних серверів, готових взяти на себе обслуговування клієнтів у випадку відмови основного сервера. Активно-активна реплікація розподіляє навантаження між кількома активними серверами, забезпечуючи як відмовостійкість, так і балансування навантаження [17].

Міграція сесій між серверами (Session Migration) дозволяє динамічно переносити ігрові сесії з одного сервера на інший без переривання ігрового процесу, що є особливо цінним для тривалих ігрових сесій та ігор з постійними світами. Цей підхід вимагає ретельної організації процесу передачі стану та механізмів перенаправлення клієнтських з'єднань.

Для забезпечення відмовостійкості в умовах тимчасових проблем з мережею застосовуються методи буферизації та черг повідомлень. Повідомлення, що не можуть бути доставлені негайно, зберігаються в буфері та надсилаються пізніше при відновленні з'єднання. Механізми гарантованої доставки (Reliable Delivery) забезпечують підтвердження отримання важливих повідомлень та автоматичну повторну передачу у випадку втрати [2].

Особливого значення набуває градація методів синхронізації за критичністю даних. Критичні дані, такі як результати бойових дій або економічні транзакції, вимагають надійної синхронізації з підтвердженням, тоді як некритичні дані, наприклад, анімації чи декоративні ефекти, можуть

синхронізуватися з використанням ненадійних протоколів для економії ресурсів.

Окрім технічних аспектів синхронізації та відмовостійкості, важливим є правильне проектування ігрової механіки з урахуванням мережевих особливостей. Дизайн толерантний до затримок (Latency-Tolerant Design) передбачає розробку ігрових механік, що залишаються приємними для користувача навіть в умовах значних затримок [6]. Це може включати розширені часові вікна для реакції, превентивні анімації для маскування затримок та адаптивні рівні складності.

На поточний момент тенденції в забезпеченні синхронізації та відмовостійкості включають застосування машинного навчання для прогнозування дій користувачів та оптимізації мережевих параметрів. Нейронні мережі можуть бути навчені передбачати найбільш ймовірні дії гравця в конкретних ситуаціях, що дозволяє зменшити відчуття затримки. Адаптивні алгоритми компресії та пріоритезації даних дозволяють оптимізувати використання пропускної здатності мережі відповідно до поточних умов з'єднання.

2.3 Алгоритми оптимізації мережевого трафіку

Проблема оптимізації мережевого трафіку набуває особливої актуальності з огляду на необхідність забезпечення комфортного геймплею для користувачів з різною якістю мережевого з'єднання та різними характеристиками апаратного забезпечення. Тому розробники ігор з мультиплеєрних механіками повинні завжди мають мати це на увазі.

Ефективна оптимізація мережевого трафіку базується на кількох основних принципах, що включають компресію даних, пріоритезацію пакетів, зменшення надлишковості інформації, а також використання специфічних методів, орієнтованих на особливості ігрового процесу. Беручи до уваги, важливий аспект того, що потрібно враховувати обмеження

пропускної здатності каналів зв'язку та вимоги до затримок у передачі даних, що є критичними для інтерактивних застосунків, розробка комплексної системи оптимізації трафіку становить важливе, та складне завдання.

Аналіз існуючих підходів до оптимізації мережевого трафіку свідчить про наявність широкого спектру методів та алгоритмів, ефективність яких залежить від конкретних умов застосування та характеристик мультиплеєрної системи.

Першим можна звернути до уваги, це методи стиснення даних, що дозволяють суттєво зменшити обсяг інформації, яка передається мережею. Стандартні алгоритми стиснення, такі як ZLIB, LZ4 або Huffman-кодування, знаходять широке застосування в ігрових застосунках [11]. Проте специфіка ігрового контенту вимагає розробки спеціалізованих підходів, що враховують семантику даних. Наприклад, для оптимізації передачі стану ігрового світу можна застосовувати дельта-компресію, яка передбачає відправлення лише змін відносно попереднього стану, а не повного набору даних [2, 14].

Ключовим елементом оптимізації мережевого трафіку є також розробка систем пріоритезації даних. У цьому контексті доцільно розрізняти критичні та некритичні дані. До критичних даних відносяться, наприклад, інформація про дії користувача, що безпосередньо впливають на геймплей, стан ключових ігрових об'єктів тощо. Некритичні дані можуть включати інформацію про візуальні ефекти, звуковий супровід, стан неігрових персонажів на периферії уваги гравця тощо. Застосування диференційованого підходу до передачі різних типів даних дозволяє ефективно розподіляти обмежені мережеві ресурси та забезпечувати належну якість обслуговування для критичних аспектів гри [3].

Значний потенціал для оптимізації мережевого трафіку мають також алгоритми інтерполяції та екстраполяції станів ігрових об'єктів. Суть цих методів полягає у прогнозуванні стану об'єктів на основі попередніх даних та фізичних моделей їх поведінки [2]. Наприклад, замість частої передачі

точного положення об'єкта, що рухається за передбачуваною траєкторією, можна передавати лише ключові параметри руху (початкова позиція, напрямок, швидкість) та оновлювати їх лише при суттєвих змінах. Клієнтська частина застосунку використовує отримані параметри для обчислення проміжних станів об'єкта (рисунок 2.5). Такий підхід дозволяє суттєво зменшити обсяг трафіку, проте вимагає розробки досить точних моделей поведінки об'єктів та механізмів корекції помилок при відхиленні фактичного стану від прогнозованого.

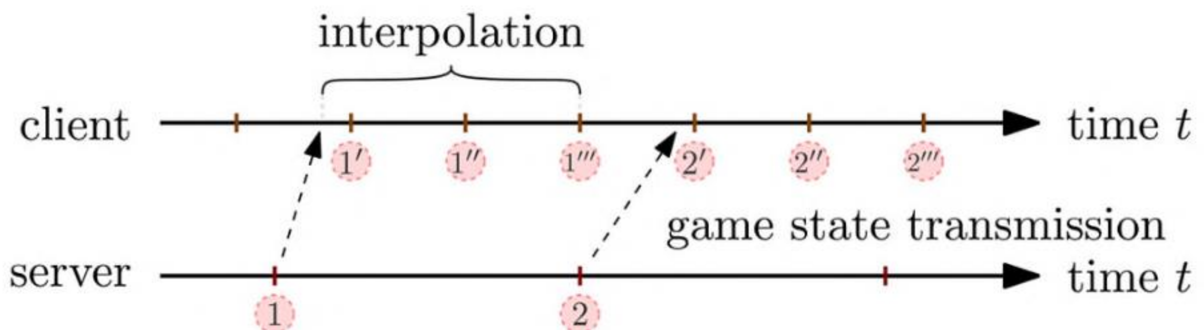

Рисунок 2.5 – Момент інтерполяції на стороні клієнта

Також є методи агрегації мережевих пакетів як ефективний засіб оптимізації трафіку. Замість відправлення окремих пакетів для кожної події або оновлення стану об'єкта, доцільно групувати дрібні оновлення в один пакет, що зменшує накладні витрати на заголовки пакетів та службову інформацію.

Додатковим напрямком оптимізації мережевого трафіку є також застосування методів просторової обмеженості. Даний підхід базується на спостереженні, що гравці зазвичай взаємодіють з обмеженою частиною ігрового світу та об'єктами, розташованими поблизу. Відповідно, можна зменшити обсяг даних, що передаються, шляхом фільтрації інформації за критерієм просторової близькості до гравця та його області інтересів. Наприклад, оновлення стану віддалених об'єктів можна передавати з меншою частотою або з нижчою деталізацією. Крім того, можна застосовувати

динамічну зміну частоти оновлень залежно від характеру ігрової активності.

Важливою частиною оптимізації мережевого трафіку є також розробка адаптивних алгоритмів, що враховують поточний стан мережевого з'єднання. Такі алгоритми дозволяють динамічно корегувати обсяг та характер даних, що передаються, відповідно до доступної пропускної здатності, затримок та інших характеристик мережі. Наприклад, при погіршенні якості з'єднання система може автоматично збільшити рівень компресії даних, знизити частоту оновлень некритичних об'єктів, пріоритезувати передачу найважливішої інформації тощо. Реалізація таких адаптивних механізмів вимагає розробки ефективних алгоритмів оцінки стану мережі та прийняття рішень щодо оптимізації трафіку в режимі реального часу.

Слід також звернути увагу на методи зменшення надлишковості даних, що передаються мережею. Одним з підходів є використання інкрементальних оновлень, при яких передаються лише зміни у стані ігрового світу, а не повний опис [9]. Інший підхід передбачає кешування даних на клієнтській стороні та використання референсних ідентифікаторів для посилання на вже відомі об'єкти замість повторної передачі їх опису. Але для ефективної реалізації таких методів необхідно розробити надійні механізми синхронізації стану між сервером та клієнтами, що забезпечують узгодженість локальних кешів та можливість відновлення при втраті пакетів.

Для мультиплеєрних ігор з великою кількістю користувачів актуальним є також питання оптимізації трансляції подій між гравцями. Традиційний підхід, що передбачає широкомовну розсилку всіх подій усім гравцям, є неефективним з точки зору використання мережевих ресурсів. Альтернативою є застосування методів фільтрації подій за критерієм значущості для конкретного гравця. Наприклад, події, що відбуваються поза зоною видимості або зоною інтересів гравця, можуть не передаватися взагалі або передаватися з нижчим пріоритетом та меншою частотою.

2.4 Виявлення та обробка мережевих затримок

Ефективна взаємодія між користувачами в ігрових застосунках значною мірою залежить від якості мережевого з'єднання. Мережеві затримки є невід'ємною частиною будь-якої онлайн-гри та суттєво впливають на ігровий процес, особливо у швидкісних жанрах, де реакція системи має бути миттєвою.

Мережева затримка визначається як час, необхідний для передачі пакета даних від клієнта до сервера і назад. В ігрових системах прийнято вимірювати затримку показником RTT (Round-Trip Time), що відображає повний цикл проходження даних (рисунок 2.6). Затримки можуть виникати з різних причин, включаючи фізичну відстань між серверами та клієнтами, перевантаження мережі, обмеження пропускної здатності та інші фактори інфраструктури.

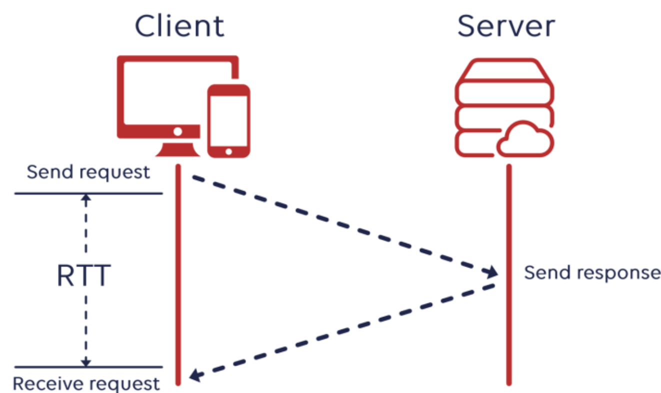

Рисунок 2.6 – Round-Trip Time

Процес виявлення мережевих затримок у сучасних ігрових системах базується на декількох методологічних підходах. Найпоширенішим є метод активного зондування, що передбачає відправлення спеціальних пакетів-зондів з часовими мітками. Клієнт надсилає такий пакет на сервер, а сервер відповідає, включаючи часову мітку отримання пакета. Після повернення пакета клієнт може розрахувати RTT як різницю між поточним часом та часом відправлення. Для підвищення точності вимірювань використовується

усереднення кількох послідовних вимірювань, що дозволяє мінімізувати вплив випадкових коливань у мережі.

Альтернативним підходом є вимірювання, що базується на аналізі часу проходження звичайних ігрових пакетів без використання спеціальних зондів. Такий метод не створює додаткового навантаження на мережу, проте потребує більш складної системи часових міток та синхронізації годинників між клієнтами та сервером.

Важливим також є класифікація затримок. Постійні, зумовлені фізичною відстанню та особливостями маршрутизації, відрізняються від тимчасових, спричинених перевантаженням мережі чи проблемами на окремих маршрутизаторах. Розуміння типу затримки дозволяє застосовувати оптимальні стратегії компенсації.

Для обробки мережевих затримок у мультиплеєрних іграх застосовують різні техніки та алгоритми, серед яких найбільш поширеними є:

- інтерполяція та екстраполяція;
- адаптація часової затримки;
- буферизація вхідних даних;
- локальний відкат (local rollback) – механізм, що дозволяє повернутися до попереднього стану гри при виявленні розбіжностей між прогнозованим та фактичним станом. Це забезпечує узгодженість ігрового світу за рахунок перерахунку подій з моменту розбіжності.

Для ефективного функціонування цих механізмів необхідна детальна діагностика мережевого з'єднання. Сучасні підходи включають безперервний моніторинг якості з'єднання з використанням показників RTT, джиттера (варіативності затримки) та втрати пакетів. На основі цих даних система може динамічно адаптувати параметри мережевого коду, наприклад, змінюючи частоту відправки оновлень, розмір пакетів або стратегію компресії даних.

Особливої уваги заслуговує проблема виявлення аномальних затримок,

що можуть сигналізувати про проблеми з'єднання або навіть спроби зловмисного втручання. Для їх виявлення застосовуються статистичні методи аналізу часових рядів, що дозволяють відокремити природні коливання затримки від підозрілих аномалій.

У контексті сучасних мережевих ігор значної популярності набули гібридні підходи до обробки затримок, що поєднують кілька технік залежно від типу гри та конкретних умов. Наприклад, для швидкісних шутерів від першої особи (FPS) часто використовують комбінацію клієнтського прогнозування та серверної компенсації затримки, тоді як для масових онлайн-ігор (MMO) більш характерні методи, орієнтовані на узгодженість ігрового світу, навіть за рахунок миттєвої реакції.

Окремим аспектом проблеми є обробка критичних подій, таких як влучення у ціль чи зіткнення об'єктів, де неточності, спричинені затримками, можуть суттєво вплинути на ігровий процес. Для таких випадків розроблено спеціалізовані алгоритми реконсиліації, що забезпечують справедливе розв'язання конфліктних ситуацій з урахуванням мережевих умов усіх залучених гравців [14].

Сучасні дослідження в цій галузі спрямовані на розробку адаптивних алгоритмів, що здатні оптимально функціонувати в різноманітних мережевих умовах, забезпечуючи найкращий компроміс між реактивністю системи та узгодженістю ігрового досвіду для всіх користувачів.

Використання штучного інтелекту та машинного навчання відкриває нові перспективи для прогнозування мережевої поведінки та адаптації системи до індивідуальних особливостей з'єднання кожного користувача.

3 АЛГОРИТМИ ОРГАНІЗАЦІЇ МЕРЕЖЕВОЇ ВЗАЄМОДІЇ В ІГРОВИХ ЗАСТОСУНКАХ

3.1 Опис проблеми у сучасних рішень

Сучасні ігрові додатки вимагають високоефективної організації мережевої взаємодії, що забезпечує швидкий обмін даними між клієнтами та сервером без надмірного навантаження на обчислювальні ресурси. Проте, під час аналізу існуючих рішень для рушія Unity, для стандартного Unity Multiplayer Netcode for GameObjects, який позиціонує себе зараз як стандарт мережевого програмування для рушія Unity, виявлено низку значних обмежень, що негативно впливають на продуктивність та масштабованість мультиплеєрних ігрових застосунків.

Основна проблема Unity Netcode полягає у неоптимальному підході до серіалізації та передачі даних. Стандартна реалізація характеризується надмірною великою кількістю інформації в пакетв даних, що створює додаткове мережеве навантаження та збільшує обсяг трафіку. Цей підхід призводить до значного зниження швидкодії при збільшенні кількості гравців або ускладненні ігрових механік. Більш того, наявна система не надає розробникам гнучких можливостей для налаштування розміру пакетів та їх пріоритезації відповідно до специфіки конкретного проекту (таблиця 3.1).

Таблиця 3.1 – Порівняльна таблиця технологій

Критерії порівняння	Unity Netcode	Розроблена система
Налаштування пакета	Unity Netcode не надає такої можливості, де при зміні Network object відправляється вся інформація	Повна кастомізація пакета для Network object
Гнучкість для Створення серверу	Є вибір між хост режимом та автономним сервером	Є вибір між хост режимом та автономним сервером
Налаштування порогу змін для відправки даних	Має свій інструмент який дозволяє відправити пакет зі змінами при певному порозі	Має можливість додати додаткову логіку валідації при зміні даних
Пріоритезації	Існує імплементації GhostField, але є проблеми з продуктивністю	Існує реалізація пріоритезації більш важливих пакетів яку розробники можуть масштабувати під себе

Критичним недоліком Unity Netcode є відсутність ефективних механізмів контролю над складом пакетів даних. Система автоматично включає до пакетів надлишкову інформацію, що часто дублюється між послідовними оновленнями стану гри. Це зумовлює неефективне використання мережевих ресурсів та створює додаткове навантаження на процесори клієнтських пристроїв при обробці отриманих даних.

3.2 Структура серверного додатка

Серверна частина мережевої взаємодії являє собою основу або ядром усієї розробляємої системи.

Розглянуто організацію мережевої взаємодії в ігрових застосунках, зосереджуючись на архітектурних особливостях серверного додатка. Серверна частина є ключовим елементом реалізації досліджуваної системи, оскільки визначає механізми обробки запитів, підтримки стану гри та взаємодії між клієнтами.

Після аналізу існуючих підходів до побудови мережевих застосунків дозволило обґрунтовано обрати клієнт-серверну модель (рисунок 3.1). Вона забезпечує ефективний розподіл навантаження, централізоване управління ресурсами та високий рівень узгодженості даних, що робить її оптимальним та гнучким рішенням для вирішення поставлених завдань під поточні задачі, які були поставленні при розробці ігрового додатку.

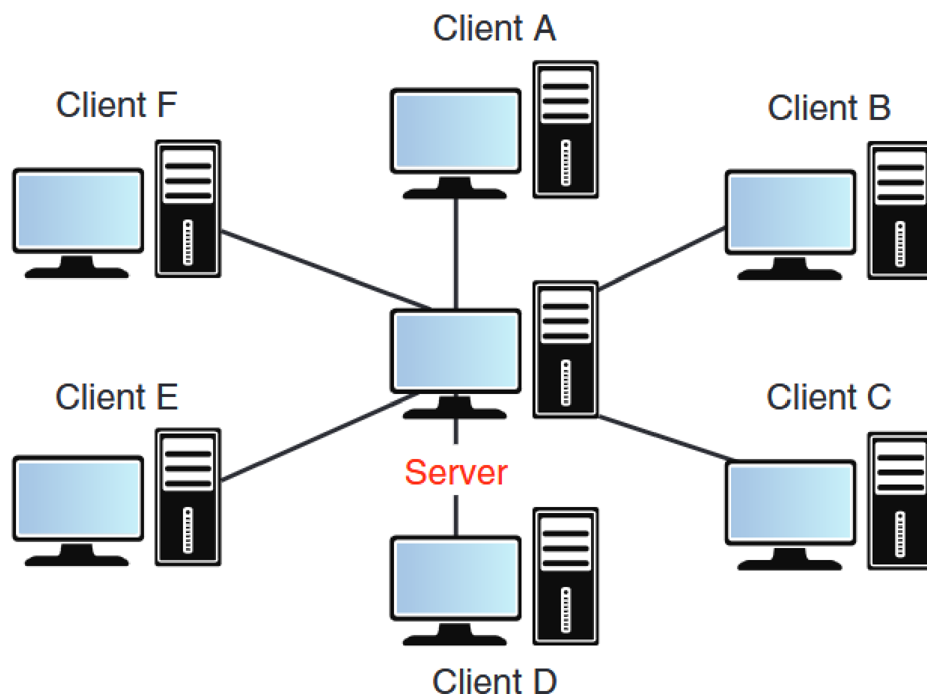

Рисунок 3.1 – Клієнт серверна модель

Гнучкість розробленої системи полягає в її здатності функціонувати у двох режимах: хост та автономний сервер. Такий підхід дозволяє оптимізувати продуктивність мережевої взаємодії залежно від вимог користувача та доступних ресурсів.

У хост-режимі, серверна і клієнтська частина гри працюють як єдина система, що забезпечує простоту запуску та мінімальні вимоги до додаткового обладнання. Серверна частина активується разом із початком ігрової сесії та припиняє роботу після її завершення. Такий варіант підходить для невеликих груп гравців або локальних багатокористувацьких ігор, оскільки зменшує потребу в окремих серверних ресурсах. Однак цей підхід має певні обмеження, наприклад таку як залежність від стабільності пристрою, хоста, що може впливати на якість з'єднання та ігровий процес.

На відміну від цього, автономний сервер працює незалежно від клієнтської частини та залишається активним, допоки його не буде примусово вимкнено (рисунок 3.2). Така архітектура дозволяє забезпечити стабільну роботу мережевого середовища, незалежно від підключених клієнтів. Сервер може бути розгорнутий на окремому пристрої або в хмарному середовищі, що дозволяє розподіляти обчислювальне навантаження та покращує загальну продуктивність. Крім того, цей підхід відкриває можливості для ведення логування, моніторингу продуктивності та застосування додаткових алгоритмів оптимізації мережевого трафіку.

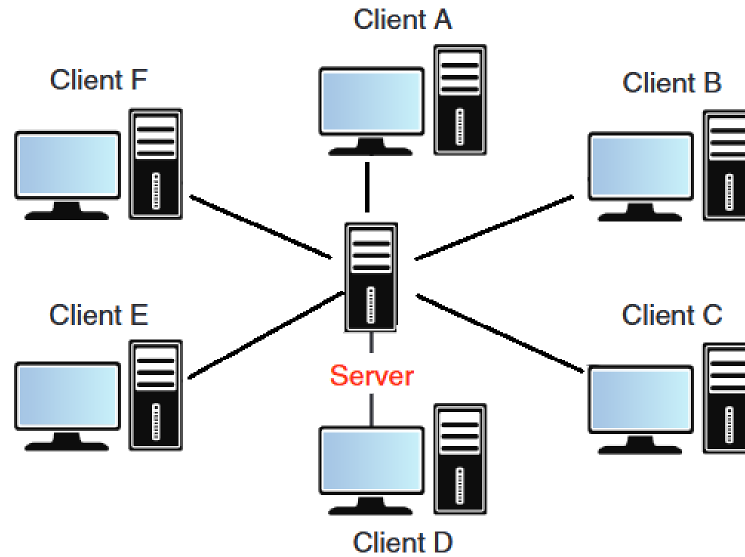

Рисунок 3.2 – Клієнт серверна частина з автономним сервером

Обидва режими забезпечують гнучкість у використанні системи, дозволяючи гравцям обирати найбільш зручний варіант залежно від їхніх потреб. Автономний сервер є оптимальним рішенням для великих ігрових спільнот або тривалих сесій із високим навантаженням на мережу, оскільки він усуває залежність від одного користувача та гарантує стабільність з'єднання. Водночас хост-режим підходить для неформальних ігрових сесій, де швидкість налаштування та простота використання мають вищий пріоритет. Таким чином, система забезпечує баланс між продуктивністю, стабільністю та зручністю використання, надаючи користувачам можливість адаптувати її під власні вимоги.

Реалізована серверна частина базується на багаторівневій архітектурі, що забезпечує структурну декомпозицію та інкапсуляцію функціональних елементів системи . Центральний серверний вузол, або ж менеджер, виконує роль координатора, що здійснює обробку запитів від множини клієнтських додатків, підтримуючи стабільний та атомарний стан ігрового середовища, а також забезпечуючи цілісність міжклієнтської взаємодії в умовах конкурентного доступу до спільних ресурсів.

Окрім цього, архітектура передбачає механізми управління потоками

даних, що мінімізують затримки та оптимізують обробку запитів навіть при високому навантаженні. Для підтримки стабільності системи впроваджено механізми, що дозволяють синхронізувати дії клієнтів у реальному часі, усуваючи розбіжності у відображенні стану гри на різних пристроях.

Таким чином, багаторівнева серверна архітектура забезпечує високу надійність, гнучкість та ефективність взаємодії між клієнтами, що є критично важливим для якісного функціонування багатокористувацького ігрового процесу.

Серверна архітектура включає підсистему валідації та фільтрації вхідних даних, що реалізує багаторівневий захист від маніпуляцій станом та інших векторів атак. Впроваджено механізми виявлення аномалій та нетипових патернів поведінки клієнтів для запобігання потенційним загрозам цілісності системи.

Значна увага в розробленій архітектурі приділена аспектам масштабованості та еластичності системи. Імплементовано модель горизонтального масштабування із застосуванням технологій контейнеризації та оркестрації, про що було також вказано вище. Такий підхід забезпечує динамічне управління обчислювальними ресурсами відповідно до актуального навантаження. Для забезпечення стабільного стану при розподілених обчисленнях впроваджено механізми синхронізації на основі розподілених блокувань та двофазного протоколу фіксації транзакцій.

Важливим компонентом серверної архітектури є підсистема обробки виключних ситуацій, що імплементує стратегію реагування на збої з можливістю швидкого повернення функціональності до робочого стану та автоматичного відновлення після критичних помилок. Така система дозволяє підтримувати та нормалізовувати ігрову сесію між гравцями з нестабільним з’єднанням, що допомагає покращити ігровий досвід. У майбутньому планується розробити механізми резервного копіювання та відновлення стану, що мінімізують потенційні втрати даних при непередбачуваних

обставинах. Це також дозволить покращити геймплейну частину застосунку.

Для моніторингу продуктивності та діагностики проблем у режимі реального часу розроблено систему логування, що забезпечує збір, агрегацію та візуалізацію ключових показників роботи серверного додатка. Імплементовано механізми автоматичного сповіщення клієнтів про критичні події та аномалії у функціонуванні системи.

Архітектурна реалізація серверного компонента враховує також аспекти енергоефективності та оптимального використання обчислювальних ресурсів через впровадження механізмів динамічного управління потоками виконання та пулами підключень до зовнішніх сервісів.

3.3 Використання бібліотек та фреймворків для мережевого програмування

Для імплементації мережевої складової системи було прийнято рішення використовувати стандартний набір бібліотек платформи .NET, та System.Net і System.Net.Sockets, що надають низькорівневий доступ до мережевих протоколів та сокетів. Такий підхід було обрано з метою забезпечення максимального контролю над процесами передачі даних та мінімізації залежностей від сторонніх компонентів, що потенційно могло б ускладнити розгортання та підтримку системи, або зробило систему обмеженою для масштабованості чи гнучкості для зміни під конкретні задачі.

Використання простору імен System.Net дозволило реалізувати роботу з мережевими ресурсами, включаючи обробку різних форматів мережевих адрес, управління мережевими підключеннями. Особливо корисними виявилися класи IPAddress та IPEndPoint для абстрагування від конкретних мережевих інтерфейсів та портів, що дозволило створити гнучку систему конфігурації серверного додатка.

Бібліотека System.Net.Sockets стала фундаментом для розробки власних класів мережевої взаємодії. На її основі було спроектовано та реалізовано

серію спеціалізованих класів, що інкапсулюють складну логіку роботи з TCP сокетами. Було розроблено абстракції для асинхронної обробки підключень, що суттєво підвищило продуктивність серверного додатка при роботі з великою кількістю одночасних клієнтських сесій.

Для забезпечення неблокуючої обробки мережевих операцій було інтегровано простір імен System.Threading.Tasks, що дозволило ефективно використовувати модель асинхронного програмування на основі задач (Task-based Asynchronous Pattern). Така архітектура дала змогу оптимізувати використання потоків та мінімізувати споживання ресурсів системи під час очікування завершення мережевих операцій.

Імплементовані класи Socket і TcpListener були розширені власними обгортками, що надають додаткові рівні абстракції та інкапсулюють складну логіку обробки мережевих подій. Було розроблено систему обробників подій, що дозволяє організувати код для реагування на різні мережеві ситуації без надмірної вкладеності та заплутаності логіки.

При проектуванні мережевої архітектури особливу увагу було приділено аспектам безпеки передачі даних. Так, реалізовано систему шифрування для захисту чутливої інформації під час передачі мережею. Для імплементації криптографічного захисту використано класи з простору імен System.Security.Cryptography, інтегровані в розроблені мережеві компоненти.

Власна імплементація мережевих компонентів, замість використання готових фреймворків на кшталт SignalR або Photon, дозволила досягти оптимального балансу між продуктивністю, контролем над мережевим стеком та гнучкістю налаштування під специфічні вимоги ігрового застосунку. Такий підхід також дав можливість впровадити специфічні оптимізації, орієнтовані на зменшення мережевого трафіку та мінімізацію затримки, що є критичними факторами для ігрових застосунків з динамічною взаємодією в реальному часі.

Розроблені класи характеризуються високим ступенем абстракції та слабкою зв'язаністю, що забезпечує їх переносимість та можливість

повторного використання в інших проектах. Така архітектура також спрощує модифікацію та розширення функціональності системи без необхідності переписувати існуючі компоненти.

При впровадженні низькорівневого мережевого коду було застосовано техніки буферизації та пулінгу об'єктів, що дозволило суттєво знизити навантаження на систему управління пам'яттю. Також, впроваджено власний механізм пулінгу буферів для мінімізації створення великих пакетів з даними, щоб не витрачати додаткові ресурси.

3.4 Реалізація основних компонентів

Архітектура розробленої системи має чітку модульну структуру, що забезпечує гнучкість, масштабованість та підтримуваність коду. Загальна архітектура мережевої підсистеми логічно розділена на кілька функціональних рівнів, що відповідає сучасним принципам багаторівневого проектування розподілених систем. Таке розділення забезпечує інкапсуляцію внутрішньої логіки кожного рівня та мінімізацію залежностей між компонентами, що значно спрощує тестування, налагодження та подальшу модифікацію системи. Основу системи становить ядро, що включає набір класів, відповідальних за організацію та управління мережевою взаємодією.

Центральним компонентом архітектури є клас NetworkManager, який виступає в ролі диспетчера та координатора всіх мережевих операцій, реалізуючи паттерн Mediator для забезпечення слабкої зв'язаності між компонентами системи. Цей компонент відповідає за ініціалізацію мережевої підсистеми, налаштування параметрів з'єднання та координацію взаємодії між іншими компонентами системи. NetworkManager реалізує паттерн Singleton, що забезпечує єдину точку доступу до мережевої функціональності для всіх компонентів ігрового застосунку. Він містить посилання на інші ключові компоненти системи та координує їх роботу відповідно до поточного стану мережі. Можна додати, що цей компонент реалізує

асинхронну модель обробки запитів, що дозволяє ефективно обслуговувати велику кількість одночасних клієнтів без блокування основного потоку виконання. Застосування асинхронної моделі обробки запитів з використанням Task-based Asynchronous Pattern (TAP) забезпечує оптимальне використання процесорного часу та мінімізацію затримки відповіді.

Для управління підключеннями використовується спеціалізований клас ConnectionManager, який інкапсулює логіку встановлення, підтримки та завершення з'єднань. Цей компонент відповідає за моніторинг стану всіх активних з'єднань, їх автентифікацію та авторизацію, а також за обробку подій підключення та відключення клієнтів. ConnectionManager реалізує механізми визначення "живих" з'єднань, видалення неактивних клієнтів та забезпечує ефективне управління ресурсами системи для оптимізації продуктивності при високих навантаженнях.

У реалізації ConnectionManager застосовується адаптивний алгоритм визначення неактивних підключень, що динамічно коригує частоту перевірок та таймаути відповідно до поточного навантаження системи та якості мережевого з'єднання (рисунок 3.3). Такий підхід дозволяє досягти оптимального балансу між швидкістю виявлення втрачених з'єднань та накладними витратами на перевірки, що особливо важливо в умовах обмежених ресурсів.

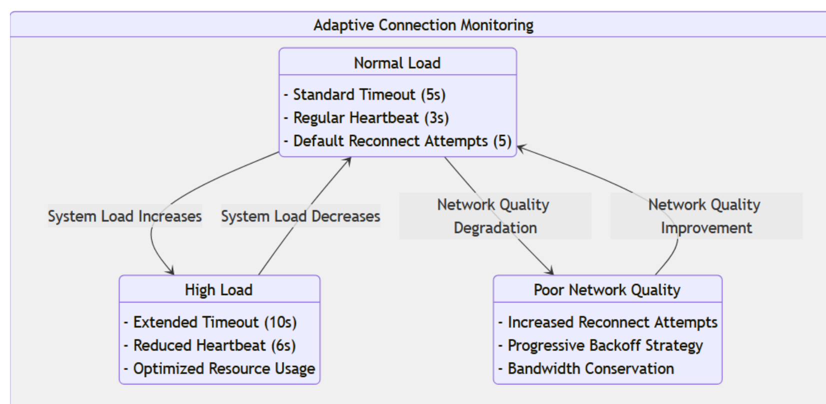

Рисунок 3.3 – Структура алгоритму визначення неактивних підключень

Серверна частина представлена класом NetworkServer, який імплементує логіку обробки вхідних підключень та повідомлень. Цей компонент реалізує логіку старту нового серверу та його зупинку. Основною роботою цього класу є зчитування та обробка даних від клієнтів або відправка даних. Також для підтримки консистентного стану гри для всіх клієнтів, цей клас зберігає у себе масив даних про клієнтів та делегує їх максимальну кількість для нормалізації роботи.

Ключовим елементом системи є клас NetworkObject, який забезпечує прозору синхронізацію стану ігрових об'єктів між сервером та клієнтами, реалізуючи патерн Proxy для віддаленого доступу до об'єктів. Цей компонент реалізує механізми серіалізації стану, фільтрації змін та оптимізованої передачі даних. NetworkObject інтегрується з ігровими об'єктами, автоматично відстежуючи зміни їх стану та забезпечуючи реплікацію цих змін на всі підключені клієнти.

Для ефективної передачі даних розроблено систему повідомлень на основі класу NetworkMessenger. Цей компонент відповідає за формування, серіалізацію та десеріалізацію мережевих пакетів (рисунок 3.5). Розроблена система використовує оптимізований бінарний формат для мінімізації розміру пакетів та зменшення мережевого трафіку. NetworkMessenger взаємодіє з абстрактним класом MessageData (рисунок 3.4), який визначає базовий інтерфейс для всіх типів повідомлень згідно з патерном Command.

Також можна додати, що для підвищення продуктивності передачі даних, таких як передача великих масивів даних наприклад інформація про ігровий світ, можна використовувати розроблену технологію стиснення даних на основі RLE алгоритму [16]. Додатково для більш прискореного та більш зручного варіанту, для створення власного мережевого пакету можна застосувати можливості ігрового рушія [17].

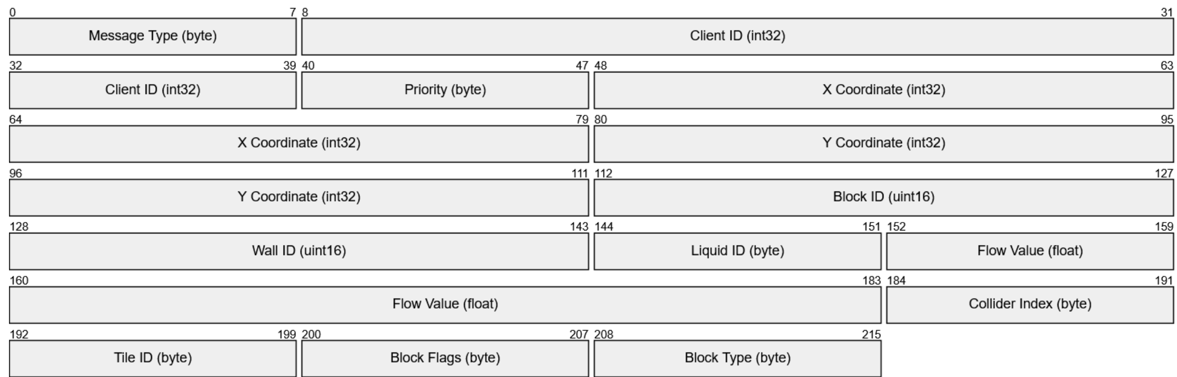

Рисунок 3.4 – Приклад пакету

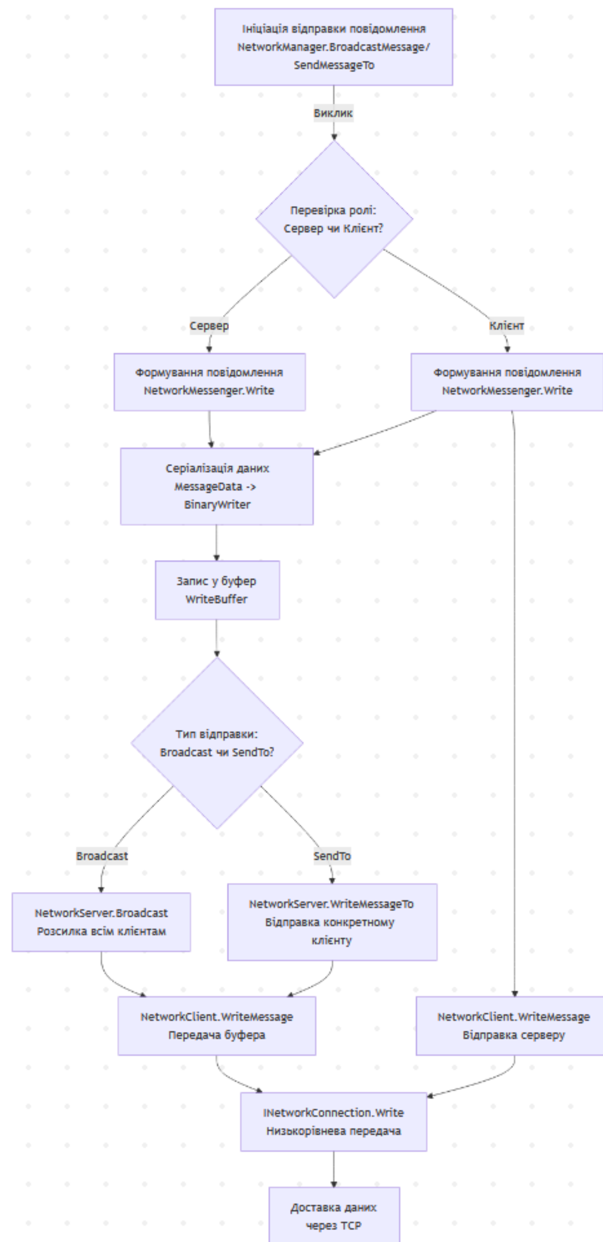

Рисунок 3.5 – Блок схема відправки даних

Імплементація NetworkMessenger включає механізми фрагментації та об'єднання пакетів для ефективної передачі даних різного обсягу (рисунок 3.6). Великі повідомлення, розмір яких перевищує максимальний розмір пакету для даного транспортного протоколу, автоматично розбиваються на фрагменти з подальшим збиранням на стороні отримувача. Навпаки, маленькі повідомлення об'єднуються в один пакет для зменшення накладних витрат на заголовки протоколів.

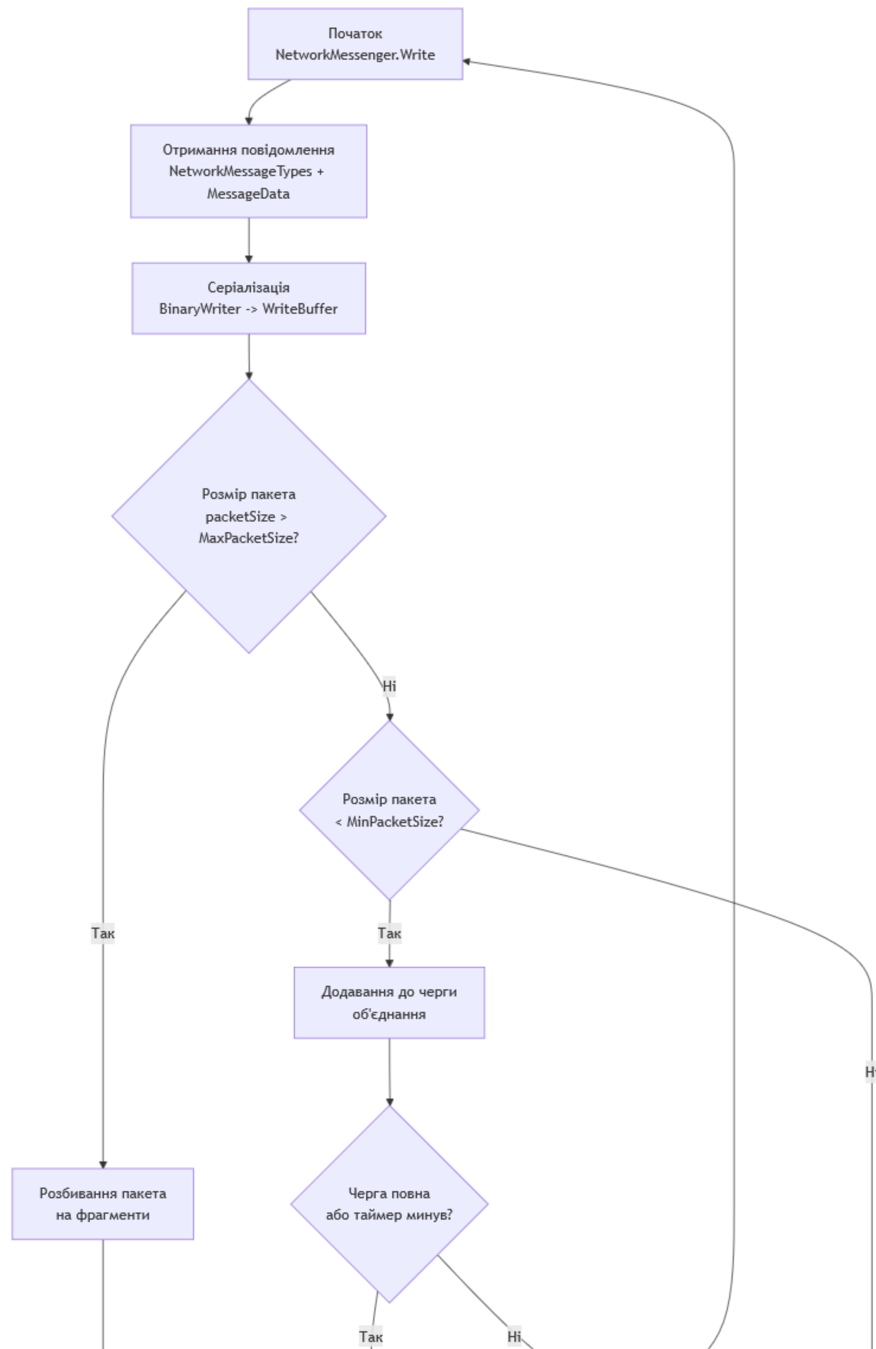

Рисунок 3.6 (а) – Механізми фрагментації та об'єднання пакетів

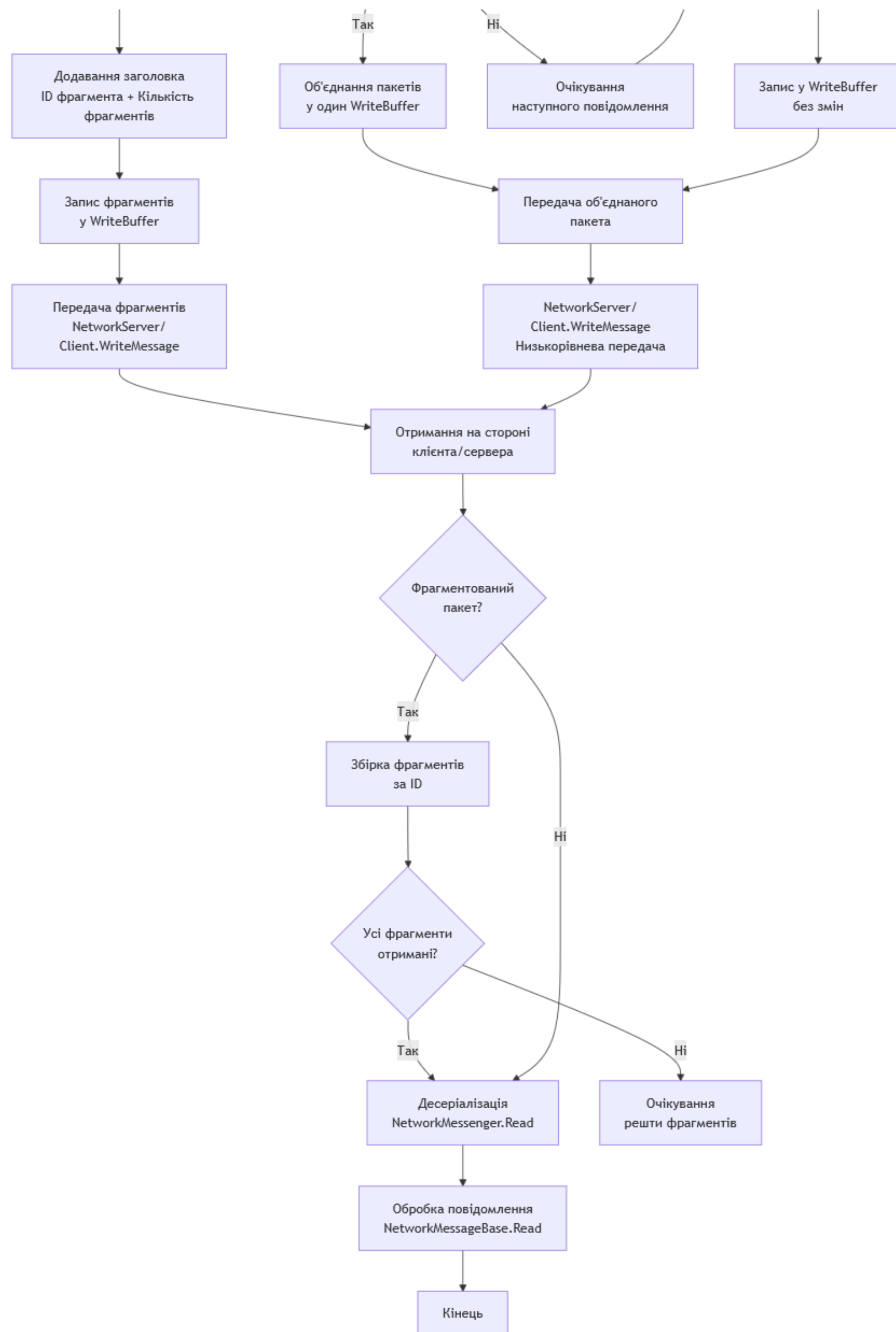

Рисунок 3.6 (б) – Механізми фрагментації та об'єднання пакетів

Для забезпечення надійності передачі даних у NetworkMessenger реалізовано алгоритм вибіркового повторення передачі (Selective Repeat ARQ), що дозволяє ефективно відновлювати втрачені пакети без необхідності повторної відправки всіх даних. Цей підхід забезпечує оптимальний баланс між надійністю та ефективністю використання мережевих ресурсів.

На основі абстрактного класу MessageData створено набір спеціалізованих класів для різних типів мережевих повідомлень. Ці класи включають повідомлення про стан ігрової сесії, створення нових мережевих об'єктів, повідомлення про відключення клієнтів, а також повідомлення, що містять ігрові дані, такі як чанки, блоки та часові мітки, пріоритет. Кожен тип повідомлення реалізує специфічну логіку серіалізації та десеріалізації, оптимізовану для конкретного виду даних.

У системі повідомлень впроваджено механізм пріоритизації та балансування навантаження, що дозволяє забезпечити першочергову обробку критичних повідомлень (таких як керуючі команди гравця) перед менш важливими даними (такими як оновлення віддалених об'єктів фону) (рисунок 3.7). Цей підхід суттєво покращує відгук системи та зменшує суб'єктивне сприйняття затримки гравцями.

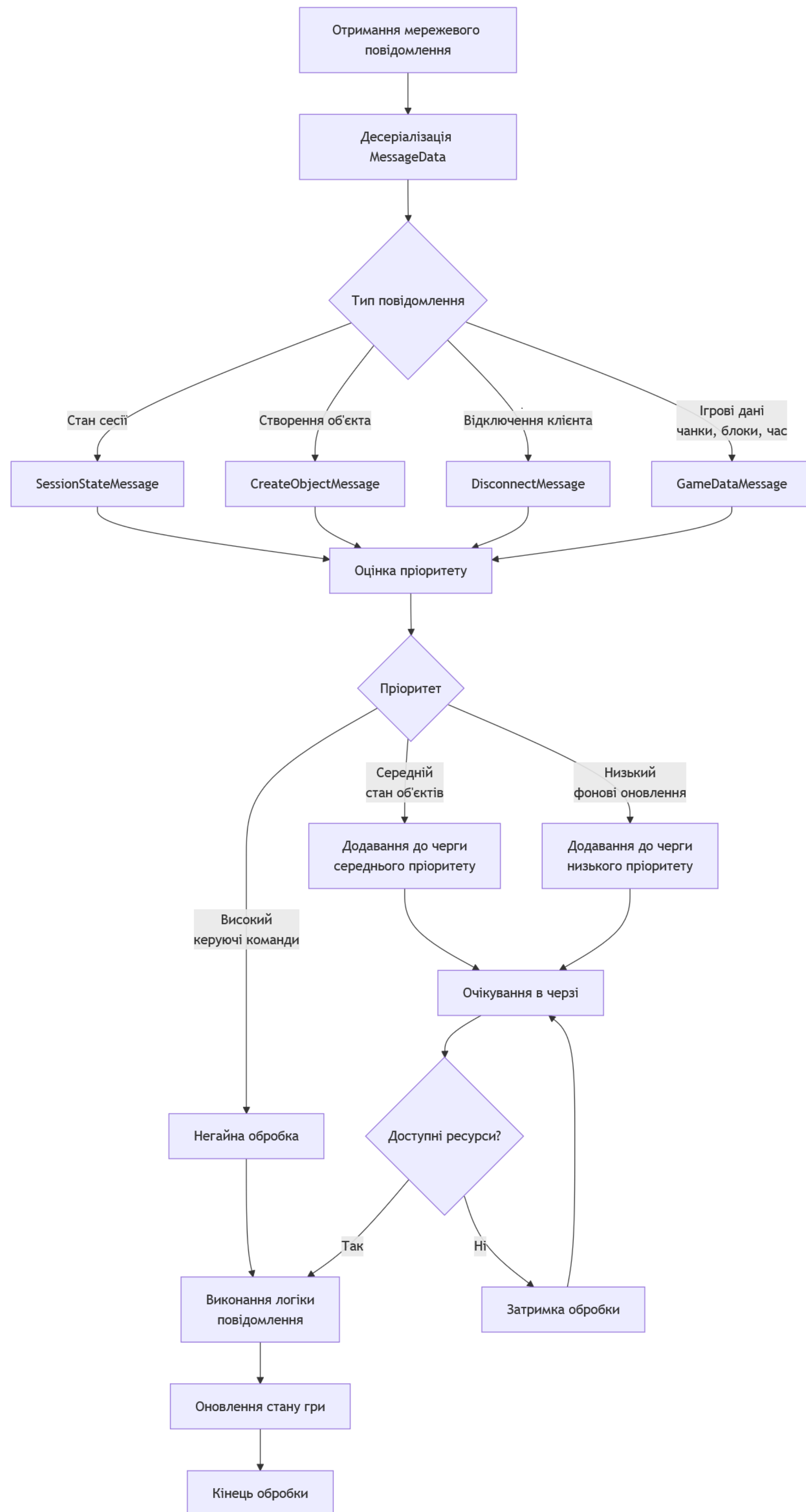

Рисунок 3.7 – Блок схема алгоритму пріоритизації

Для забезпечення транспортного рівня мережевої взаємодії було розроблено інтерфейс INetworkConnection та його конкретну імплементацію NetworkConnection, що забезпечує роботу з TCP протоколом. Використання інтерфейсу дозволяє досягти гнучкості системи та можливості її розширення для підтримки різних транспортних протоколів без необхідності зміни інших компонентів системи. Така архітектура забезпечує можливість додавання нових реалізацій для підтримки UDP, WebSockets або інших протоколів при необхідності.

Клас NetworkConnection реалізує низькорівневу взаємодію з мережевим стеком операційної системи, інкапсулюючи деталі встановлення та підтримки з'єднання, буферизації даних та обробки помилок. Для оптимізації продуктивності в цьому компоненті застосовуються техніки неблокуючого вводу-виводу (non-blocking I/O) та асинхронні операції, що дозволяють ефективно обробляти велику кількість підключень без створення надмірної кількості потоків.

Додатково в NetworkConnection реалізовано механізми адаптивного управління потоком даних (flow control), що динамічно регулюють швидкість передачі відповідно до доступної пропускної здатності мережі та поточного навантаження на приймаючу сторону (таблиця 3.2). Це запобігає перевантаженню мережі та втраті пакетів, що особливо важливо для стабільної роботи в умовах обмеженої або непостійної пропускної здатності.

Таблиця 3.2 – Порівняльна таблиця NetworkConnection між технологіями

Характеристика	Розроблена система (INetworkConnection/TCPNetworkConnection)	Unity Netcode NetworkConnection
1	2	3
Архітектура	Модульна система з інтерфейсом INetworkConnection та конкретною реалізацією TCPNetworkConnection	Монолітна система, тісно інтегрована з іншими компонентами Unity Netcode
Протоколи	Гнучка архітектура з можливістю підключення різних протоколів (TCP, UDP, WebSockets) через реалізацію інтерфейсу	Обмежена розширюваність, залежність від транспортних систем Unity
Оптимізація даних	Оптимізована передача даних з мінімальними заголовками, що збільшує швидкість передачі	Стандартна серіалізація Unity з додатковими метаданими
Асинхронність	Повністю асинхронна архітектура з використанням Task і неблокуючого вводу-виводу	Змішана модель з використанням корутин Unity
Керування з'єднаннями	Система автоматичного підключення/відключення клієнтів з обробкою подій	Система управління з'єднаннями через NetworkManager
Буферизація	Ефективна буферизація з оптимізованим розміром повідомлень	Стандартна буферизація з фіксованими розмірами буферів

Продовження таблиці 3.2

1	2	3
Обробка помилок	Розвинена система обробки помилок з логуванням та відновленням	Базова система обробки помилок
Управління потоком даних	Адаптивне управління потоком даних залежно від навантаження	Статичне управління потоком даних
Управління станом	Інтеграція з State Machine для автоматичного керування станами мережі	Ручне управління станами через API
Надійність	Складна система черговості повідомлень для забезпечення надійності	Вбудована система надійності з обмеженою конфігурацією
Масштабованість	Розроблена для ефективної роботи з великою кількістю з'єднань	Оптимізована для типових ігрових сценаріїв з обмеженою кількістю гравців
Моніторинг	Комплексна система моніторингу стану з'єднань через інтеграцію зі State Machine	Базовий моніторинг через події і колбеки, Profiler
Налаштування	Високий рівень налаштування параметрів з'єднання (буфер, таймаути, порти)	Високий рівень налаштування параметрів з'єднання

Продовження таблиці 3.2

1	2	3
Інтеграція з рештою системи	Чітка сегментація відповідальності через інтерфейси та шаблон проектування Singleton	Тісна інтеграція з компонентами Unity через наслідування MonoBehaviour або через проектування Singleton
Додатковий функціонал	Включає власні рішення для серіалізації, стиснення даних та шифрування	Використовує стандартні рішення Unity для серіалізації та захисту

3.5 Модель синхронізації об’єктів гри

У рамках реалізації мережевої архітектури розробленого ігрового застосунку було приділено час для створення моделі синхронізації об'єктів гри. Ефективна синхронізація стану ігрових об'єктів є критичним компонентом для забезпечення узгодженого ігрового досвіду між різними клієнтами в багатокористувацькому середовищі.

Розроблена модель синхронізації ґрунтується на архітектурі з чітким розподілом відповідальності між компонентами, що дозволяє досягти балансу між продуктивністю та точністю відтворення стану гри. В основі даної моделі лежить система об'єктів, яка забезпечує відстеження, створення, оновлення та знищення ігрових сутностей у мережевому просторі.

Центральним елементом реалізованої системи синхронізації є клас NetworkObject, який виступає базовим компонентом для всіх мережевих об'єктів у грі. Даний клас інкапсулює фундаментальні властивості та поведінку, необхідні для мережевої взаємодії: унікальну ідентифікацію,

позиціонування у просторі та визначення власності (приналежності) об'єкта. Кожен об'єкт у системі має два рівні ідентифікації: глобальний ідентифікатор, що визначає тип об'єкта у загальній ієрархії, та унікальний ідентифікатор екземпляра, що забезпечує можливість адресації конкретного об'єкта під час мережевої взаємодії.

Для керування колекцією мережевих об'єктів реалізовано клас NetworkObjects, який забезпечує централізоване управління життєвим циклом ігрових сутностей. Цей компонент відповідає за створення об'єктів, їх реєстрацію в системі, пошук за ідентифікатором та видалення.

NetworkTransform забезпечує синхронізацію просторових характеристик об'єкта: позиції, обертання та масштабування. Для оптимізації мережевого трафіку реалізовано механізм порогового надсилання оновлень, коли інформація про зміну позиції передається лише у випадку, якщо різниця між поточним та попереднім станом перевищує встановлений поріг. Такий підхід дозволяє значно зменшити обсяг мережевих повідомлень без істотного впливу на якість синхронізації.

NetworkAnimator відповідає за синхронізацію анімаційних станів об'єктів, забезпечуючи візуальну узгодженість дій на всіх клієнтах. Даний компонент відстежує зміни поточного стану анімації та передає відповідні дані (хеш анімації) іншим учасникам сесії. Використання хешів замість рядкових ідентифікаторів дозволяє оптимізувати обсяг даних, що передаються.

Для забезпечення коректного функціонування системи синхронізації реалізовано чіткий поділ логіки обробки для об'єктів, якими керує локальний гравець (власник об'єкта), та об'єктів, контрольованих іншими гравцями. Власники об'єктів відповідають за відстеження змін стану та їх поширення, тоді як інші приймають оновлення та застосовують їх до локальних представлень об'єктів (рисунок 3.8).

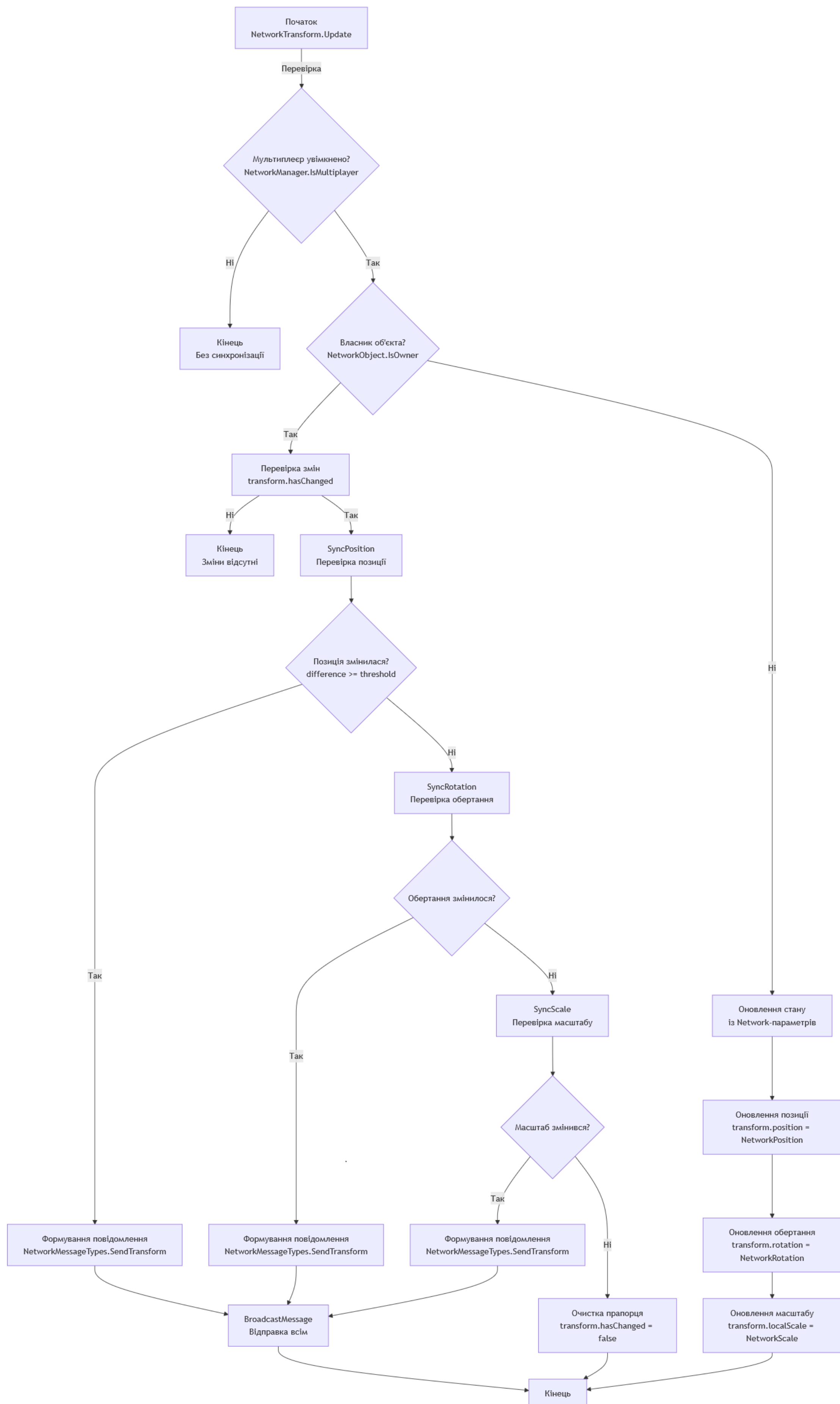

Рисунок 3.8 – Блок схема алгоритму синхронізації

Асинхронність середовища виконання вимагає особливого підходу до обробки операцій створення та знищення об'єктів. Для забезпечення безпечної роботи з ігровими об'єктами в багатопотоковому контексті використовується утилітний клас MainThreadUtility, який дозволяє відкласти виконання операцій до наступного кадру основного потоку.

Комунікаційний прошарок системи синхронізації реалізовано на основі подійно-орієнтованої моделі з використанням типізованих повідомлень (MessageData), які передаються через центральний NetworkManager. Цей підхід дозволяє абстрагувати логіку синхронізації від конкретної реалізації мережевого транспорту та забезпечити гнучку розширюваність системи.

Для оптимізації пропускної здатності мережі застосовано кілька стратегій:

- використання компактних числових типів для передачі даних (int, float, long);
- диференційована частота оновлень для різних типів даних з урахуванням їх пріоритетності;
- пакетна передача оновлень;
- порогова фільтрація змін для запобігання надлишковим повідомленням.

Важливим частиною розробленої моделі синхронізації є її адаптивність до різних умов мережевого з'єднання. Система здатна функціонувати як у режимі локальної гри, так і в мультиплеєрному середовищі, автоматично адаптуючи свою поведінку до відповідного контексту.

Для забезпечення узгодженості ігрового стану на різних клієнтах реалізовано механізм детермінованої генерації ідентифікаторів об'єктів з використанням ObjectIDGenerator. Це дозволяє гарантувати унікальність ідентифікаторів у межах сесії та забезпечити коректне посилання на об'єкти при обміні повідомленнями.

Оцінка ефективності розробленої моделі синхронізації проводилася за кількома ключовими параметрами:

- затримка синхронізації станів між клієнтами;
- обсяг мережевого трафіку;
- обчислювальне навантаження на клієнтах;
- стійкість до втрати пакетів та інших мережевих проблем.

Експериментальні дані показали, що запропонована модель забезпечує прийнятний рівень синхронізації при помірному використанні мережевих ресурсів навіть у середовищі з обмеженою пропускною здатністю.

У перспективі розвитку моделі синхронізації планується впровадження додаткових методів оптимізації, інтерполяції станів об'єктів для згладжування візуальних ефектів при втраті пакетів, а також алгоритмів передбачення для компенсації мережевої затримки. Такі вдосконалення дозволять додатково підвищити якість користувацького досвіду та розширити можливості системи для підтримки більш динамічних ігрових сценаріїв.

Розроблена модель синхронізації об'єктів гри демонструє ефективний баланс між складністю реалізації та функціональними можливостями, забезпечуючи надійну основу для створення розподілених ігрових систем з високою інтерактивністю та узгодженістю стану.

3.6 Логування та діагностика мережевих подій

Логування та діагностика є критично важливими аспектами розробки мережевого застосунку, в контексті багатокористувацьких ігор. Вони дозволяють відстежувати події, що відбуваються в мережевій взаємодії, діагностувати проблеми з продуктивністю та забезпечувати стабільність системи. У цьому розділі розглянуто реалізацію механізму логування та діагностики мережевого обміну даними в рамках створеного ігрового серверного додатка.

3.6.1 Призначення та необхідність логування

У мережевих іграх передача даних між клієнтом і сервером є безперервним процесом, що включає запити, відповіді, обмін станами об'єктів гри та синхронізацію дій гравців. Будь-які збої в цьому процесі можуть спричинити значні проблеми, такі як затримки, втрата даних або навіть відключення клієнтів. Впровадження системи логування дозволяє отримувати інформацію про кожен важливий мережевий запит і відповідь, відстежувати зміни в ігровому середовищі та аналізувати можливі причини відмови.

Окрім діагностики помилок, логування також використовується для збору статистики про навантаження на сервер, частоту запитів та їх обробку. Це допомагає виявити вузькі місця системи та оптимізувати її роботу.

3.6.2 Реалізація системи логування

У практичній частині проєкту для організації логування використовується клас GameConsole, який реалізує механізм збереження та виводу логів у текстовому форматі. Основні функції цього класу включають: додавання, очищення логів, взаємодія з користувачем.

Говорячи про перший метод, функція Log(string text) дозволяє зберігати текстові повідомлення в буфері логів. Також передбачено можливість виділення логів кольорами (Log(string text, Color color)), що допомагає у візуальному розрізненні звичайних повідомлень та помилок (LogError(string text)). Наступним йде Метод ClearText(), що дозволяє видалити всі попередні записи з лог-файлу. І наостанок, користувач може відкривати консоль розробника, переглядати логи та вводити команди для діагностики мережевих процесів.

В основі логування лежить StringBuilder, який використовується для накопичення логів, що зменшує навантаження на систему порівняно зі

збереженням у звичайному списку рядків.

3.6.3 Діагностика та аналіз логів

Логи використовуються як засіб моніторингу роботи мережевих компонентів. У логах можуть зберігатися такі події:

- підключення та відключення клієнтів;
- обмін даними між клієнтом і сервером;
- попередження про можливі проблеми з мережею, такі як підвищена затримка чи втрата пакетів.
- Фіксація критичних помилок, що можуть спричинити збій сервера.

Завдяки інтеграції логів у внутрішню консоль розробника, відстеження мережевих подій може здійснюватися без необхідності додаткових зовнішніх інструментів. Крім того, консоль дозволяє вводити команди для тестування мережевих сценаріїв у реальному часі.

3.6.4 Оптимізація та майбутнє розширення системи логування

Хоча поточна реалізація логування є достатньою для базового моніторингу, у подальшому система може бути розширена за рахунок наступних покращень:

- збереження логів у файлах дозволить аналізувати події навіть після завершення сеансу гри або перезапуску сервера;
- фільтрація логів надасть можливість виводити лише певні типи подій (наприклад, тільки помилки або тільки мережеві запити);
- автоматичний аналіз логів через впровадження алгоритмів, які зможуть визначати аномалії у мережевій взаємодії. Наприклад, раптове збільшення затримки чи відмову в обробці запитів.

Таким чином, реалізація логування та діагностики в ігрових мережевих застосунках є важливим етапом забезпечення їхньої стабільності.

4 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ТА ТЕСТУВАННЯ

4.1 Опис методології тестування мережевої взаємодії

Методологія тестування базується на системному підході до оцінки якості мережевої взаємодії та включає набір інструментальних засобів моніторингу, та аналітичні методи інтерпретації отриманих результатів. Для забезпечення точності вимірювань було розроблено спеціалізоване тестове середовище, яке емулює реальні умови експлуатації мультиплеєрної системи.

Для тестування було створено механізм навантаження на поточний контекст, що включає у собі симуляцію втрати пакетів, затримки або обмеження пропускної здатності. Це дозволило моделювати різноманітні сценарії мережевої взаємодії, від ідеальних умов локальної мережі до складних умов мобільного інтернет-з'єднання з високою затримкою та нестабільністю каналу.

Інструментарій тестування включає:

- системи моніторингу мережевого трафіку на рівні пакетів (Wireshark, tcpdump) для детального аналізу структури комунікацій;
- програмні профілювальники навантаження на серверну частину системи, що дозволяють виявити вузькі місця та оптимізувати обробку мережевих подій.

Методологія передбачає проведення серії тестів для кожного з досліджуваних підходів до організації мережевої взаємодії, з подальшим порівняльним аналізом отриманих результатів. Для кожного тесту визначено набір контрольованих умов, процедуру проведення та критерії успішності. Результати тестування фіксуються у вигляді числових показників, графіків залежностей та процентильних розподілів ключових метрик.

Створена методологія тестування дозволяє не лише оцінити поточну реалізацію мережевої підсистеми, але й прогнозувати її поведінку при

масштабуванні та зміні умов експлуатації. Комплексний характер тестування забезпечує виявлення потенційних проблем продуктивності та надійності на ранніх етапах розробки, що суттєво знижує ризики при розгортанні системи в виробничому середовищі.

4.2 Тестування продуктивності при різній кількості підключених клієнтів

Експериментальна база дослідження включала серверну платформу з характеристиками, наведеними у таблиці 4.1. Для моніторингу системних параметрів використовувався комплекс засобів, що включав інтегровані у серверний застосунок лічильники продуктивності та зовнішні інструменти профілювання (Performance Monitor, Intel VTune Profiler). Тестове навантаження генерувалося за допомогою програмних емуляторів клієнтів, які виконували типові користувацькі сценарії з контрольованою інтенсивністю дій.

Таблиця 4.1 – Характеристики тестової серверної платформи

Компонент	Характеристика
Процесор	11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz
Оперативна пам'ять	16 ГБ DDR4-3200 МГц
Мережевий інтерфейс	10 Гбіт/с Ethernet
Операційна система	Windows 11
Сховище даних	NVMe WDC PC SN530 SDBPNPZ-512G-1114

Методика проведення експериментів базувалася на покроковому нарощуванні навантаження з фіксацією усталених значень контрольованих

параметрів на кожному етапі. Тестування проводилося з різною кількістю одночасно підключених клієнтів: 2, 5, 10, 15. Для кожного рівня навантаження проводилася серія з п'яти вимірювань тривалістю 15 хвилин, що дозволило отримати статистично достовірні результати та оцінити варіабельність показників.

Результати дослідження утилізації обчислювальних ресурсів сервера представлені у таблиці 4.2, де відображено залежність завантаження процесора від кількості підключених клієнтів. Отримавши статистичні дані (рисунки 4.1-4.5), можна спостерігати динаміку змін параметрів при різній кількості клієнтів.

Таблиця 4.2 – Залежність показників продуктивності від кількості підключених клієнтів

Кількість клієнтів	Утилізація CPU (%)	Використання пам'яті (МБ)	Вихідний трафік (КБ/с)	Затримка оновлень (мс)
2	45	161	2,2	51
5	50	177	10	55
10	58	210	26	61
15	64	245	61,5	67

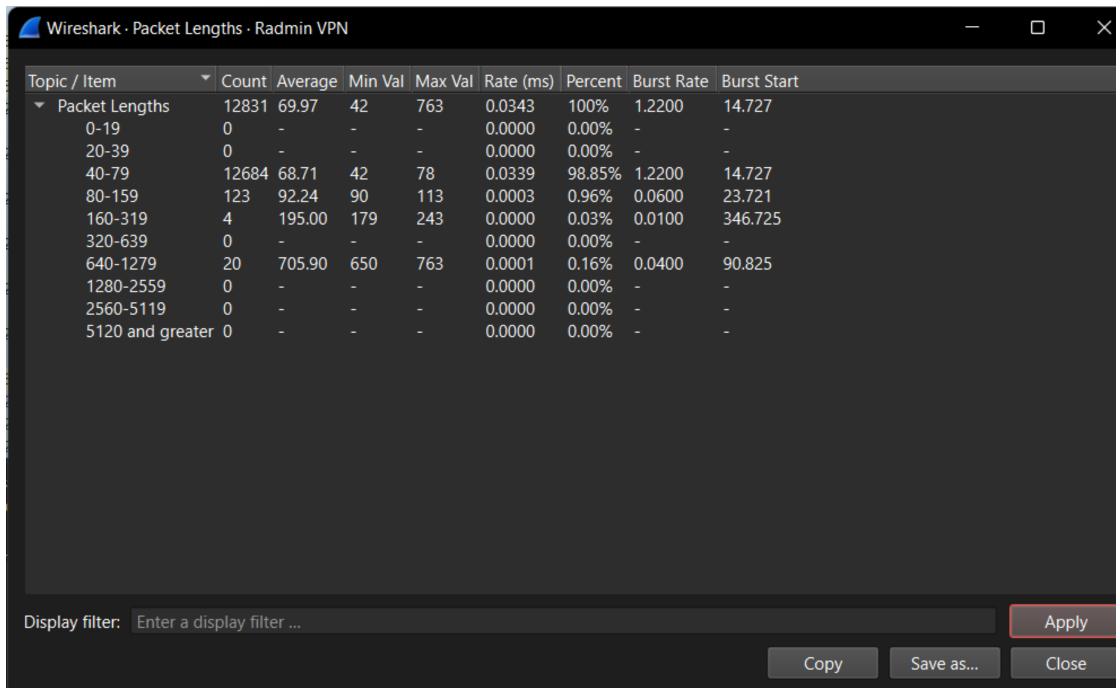

Рисунок 4.1 – Таблиця показників розміру пакетів

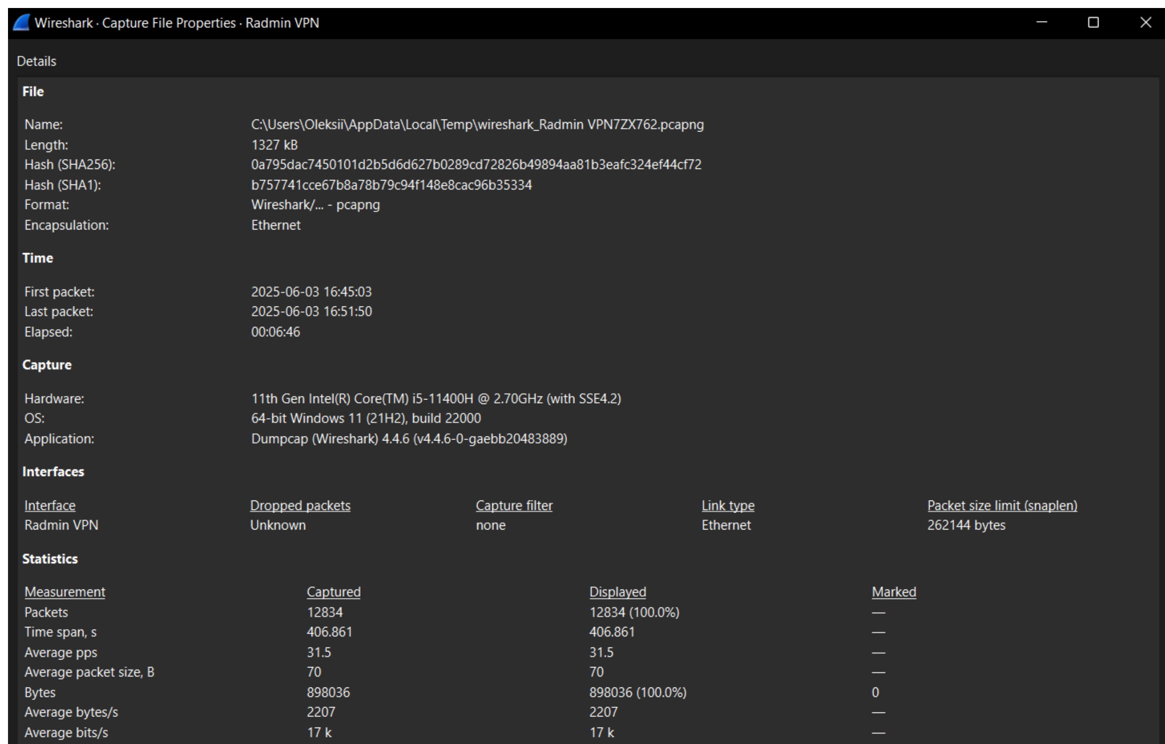

Рисунок 4.2 – Статистичні дані для 2 клієнтів

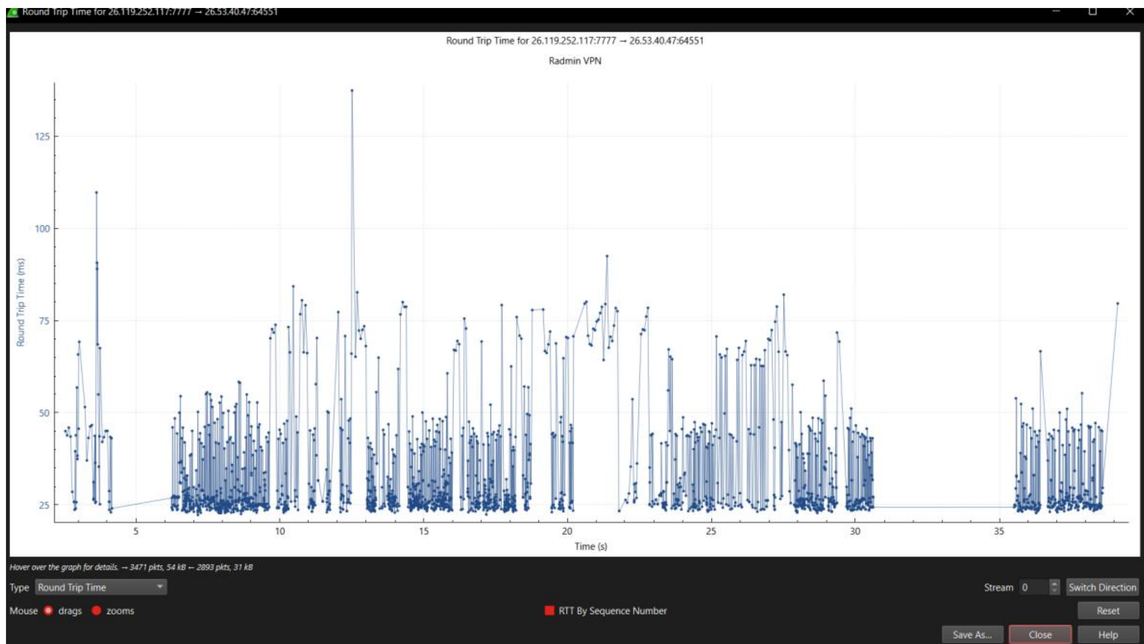

Рисунок 4.3 – Графік середньої затримки для 2 клієнтів

Wireshark · Capture File Properties · Radmin VPN

Details

File

Name:	C:\Users\Oleksii\AppData\Local\Temp\wireshark_Radmin VPNWHF262.pcapng
Length:	2216 kB
Hash (SHA256):	353401585e51e139f19e378b335065b51ed787eff77726e34bd1324187d797e1
Hash (SHA1):	e5cb23721a019a4c52ec76cee68da97fcc793912
Format:	Wireshark/... - pcapng
Encapsulation:	Ethernet

Time

First packet:	2025-06-03 17:06:23
Last packet:	2025-06-03 17:06:53
Elapsed:	00:00:29

Capture

Hardware:	11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz (with SSE4.2)
OS:	64-bit Windows 11 (21H2), build 22000
Application:	Dumpcap (Wireshark) 4.4.6 (v4.4.6-0-gaebb20483889)

Interfaces

Interface	Dropped packets	Capture filter	Link type	Packet size limit (snaplen)
Radmin VPN	0 (0.0%)	none	Ethernet	262144 bytes

Statistics

Measurement	Captured	Displayed	Marked
Packets	21604	4341 (20.1%)	—
Time span, s	29.721	29.721	—
Average pps	726.9	146.1	—
Average packet size, B	69	69	—
Bytes	1494663	299641 (20.0%)	0
Average bytes/s	50 k	10 k	—
Average bits/s	402 k	80 k	—

Рисунок 4.4 – Статистичні дані для 5 клієнтів

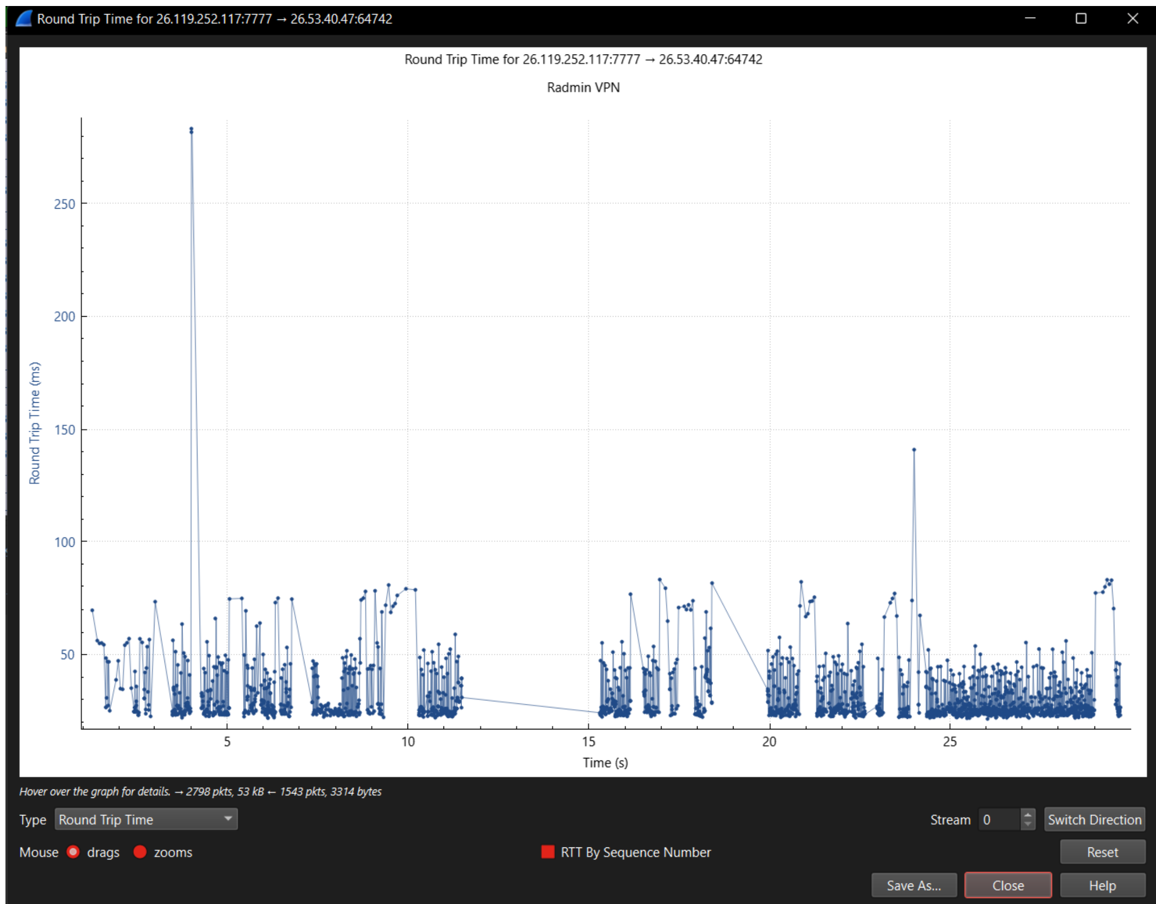

Рисунок 4.5 – Графік середньої затримки для 5 клієнтів

Спостерігається практично лінійне зростання утилізації CPU до рівня приблизно 15 клієнтів, після чого динаміка змінюється, демонструючи ознаки насичення. При 100 клієнтів загальне завантаження процесора досягає 85%, а при 150 клієнтів – 98%, що свідчить про наближення до граничної пропускної здатності системи.

Дослідження динаміки використання оперативної пам'яті показало стабільне лінійне зростання з коефіцієнтом приблизно 6,6 МБ на кожного підключеного клієнта. При максимальному тестовому навантаженні (15 клієнтів) загальне споживання пам'яті досягло 245 МБ, що знаходиться в межах доступних ресурсів тестового сервера. Аналіз структури використання пам'яті показав, що близько 40% виділяється під кеш станів об'єктів, 25% під буфери мережевих повідомлень, 20% під структури даних для пошуку релевантних об'єктів, і 15% під інші службові дані.

Критичним параметром для мультиплеєрних систем є пропускна

здатність мережевого стеку сервера. Результати вимірювань показали, що вихідний трафік зростає нелінійно відносно кількості підключених клієнтів. При 2 клієнтах загальний вихідний трафік становив близько 2,2 КБ/с, а при 15 клієнтах досяг 61,5 КБ/с. Нелінійність пояснюється зростанням кількості взаємодій між об'єктами, що потребують синхронізації, та квадратичною залежністю кількості потенційних взаємодій від числа активних об'єктів у віртуальному просторі.

Важливою характеристикою продуктивності є затримка оновлення стану гри, яка визначає суб'єктивне сприйняття швидкості відгуку системи з боку користувачів. Проведені вимірювання показали, що середня затримка залишається в прийнятних межах (до 100 мс) при навантаженні до 15 клієнтів, однак при подальшому збільшенні кількості з'єднань спостерігається її стрімке зростання, досягаючи 150 мс при 100 клієнтів. При цьому максимальні значення затримки при високих навантаженнях перевищують 260 мс, що може негативно впливати на ігровий досвід користувачів, особливо в динамічних сценаріях.

4.3 Тестування продуктивності при різному мережевому навантаженні

Швидкість мережевого з'єднання безпосередньо впливає на якість ігрового досвіду користувачів, особливо в контексті real-time взаємодії, де навіть незначні затримки можуть призвести до суттєвого погіршення геймплею.

Методологія проведення експериментів базувалася на варіюванні пропускної здатності мережевого каналу з метою моделювання різних сценаріїв підключення користувачів. Для забезпечення достовірності результатів було створено тестове середовище, що дозволило імітувати широкий спектр мережевих умов від високошвидкісних оптоволоконних з'єднань до обмежених мобільних підключень.

Перший етап тестування передбачав моделювання оптимальних

мережевих умов з пропускною здатністю 300 Мбіт/с, що відповідає сучасним стандартам локальних мереж та високошвидкісних інтернет-підключень. За таких умов система демонструвала максимальну продуктивність, характеризуючись мінімальною затримкою передачі даних та стабільним рівнем синхронізації між клієнтами. Середній час відгуку складав менше 50 мілісекунд, що забезпечувало практично миттєву реакцію на дії користувачів.

Подальше зниження пропускної здатності до 150 Мбіт/с, характерного для стандартних домашніх інтернет-підключень, виявило початкові ознаки деградації продуктивності. Спостерігалося незначне збільшення затримки до 50 мілісекунд, проте система зберігала прийнятний рівень функціональності завдяки впровадженим алгоритмам оптимізації мережевого трафіку та адаптивному управлінню частотою оновлень.

Для перевірки іншої ситуації, було знижено швидкість до 75 Мбіт/с, що відповідає умовам обмеженого широкосмугового доступу або мобільного інтернету. За таких параметрів система почала проявляти суттєві ознаки навантаження, включаючи збільшення затримок до 65 мілісекунд проте система залишалась стабільною.

Тестування в умовах низької пропускної здатності (10-20 Мбіт/с) продемонструвало граничні можливості системи адаптації. Затримка зросла до 75 мілісекунд, що все одно показало задовільний результат. Все це завдяки реалізованому механізму пріоритизації важливих пакетів. Через це система зберігала базову функціональність.

Для систематизації отриманих результатів було складено таблицю 4.2, що демонструє залежність показника затримки від параметрів мережевого з'єднання.

Таблиця 4.2 – Залежність продуктивності системи від пропускної здатності мережі

Пропускна здатність	Середня затримка	Якість синхронізації
300 Мбіт/с	40-45 мс	Відмінна
150 Мбіт/с	45-50 мс	Добра
75 Мбіт/с	60-70 мс	Задовільна
18 Мбіт/с	70-75 мс	Задовільна

Аналіз результатів тестування показав, що система демонструє нелінійну залежність продуктивності від пропускної здатності мережі. Найбільш критичним діапазоном виявився перехід від 150 до 75 Мбіт/с, де спостерігається найзначніший стрибок затримки – з 50 до 65 мілісекунд, що становить 30% збільшення показника. Це свідчить про те, що система має певний поріг оптимальної роботи, нижче якого ефективність різко знижується. Також результат можна побачити у вигляді графіків (рисунки 4.6–4.9)

Рисунок 4.6 – Середня затримка при пропускній здатності в 300 МБіт/с

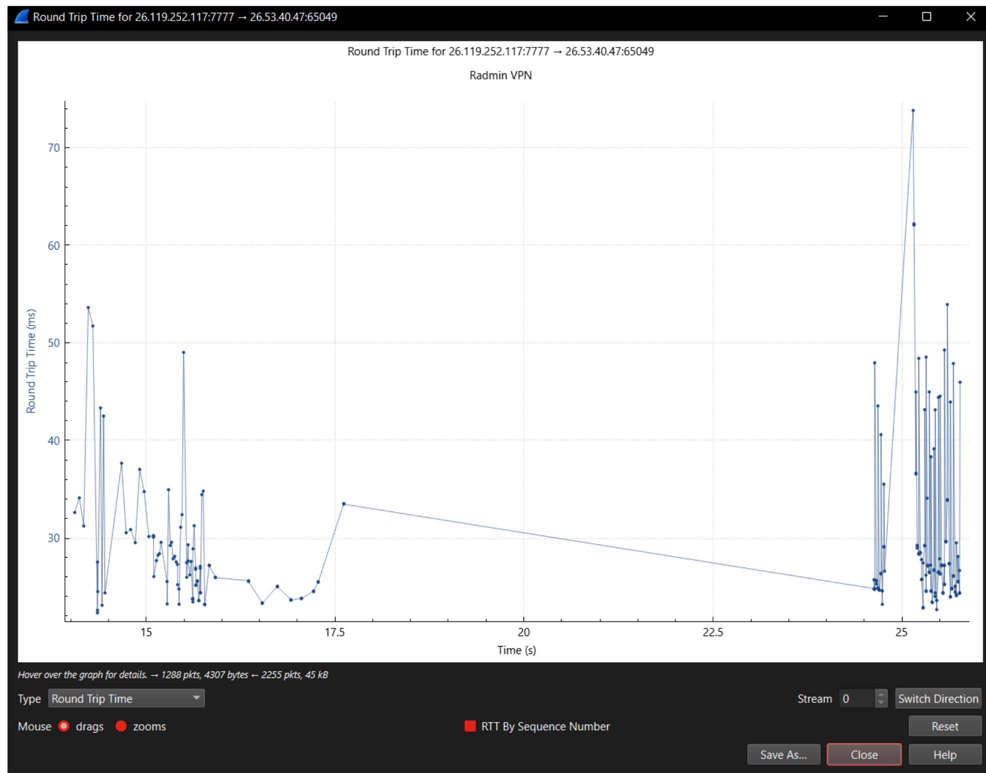

Рисунок 4.7 – Середня затримка при пропускній здатності в 150 МБіт/с

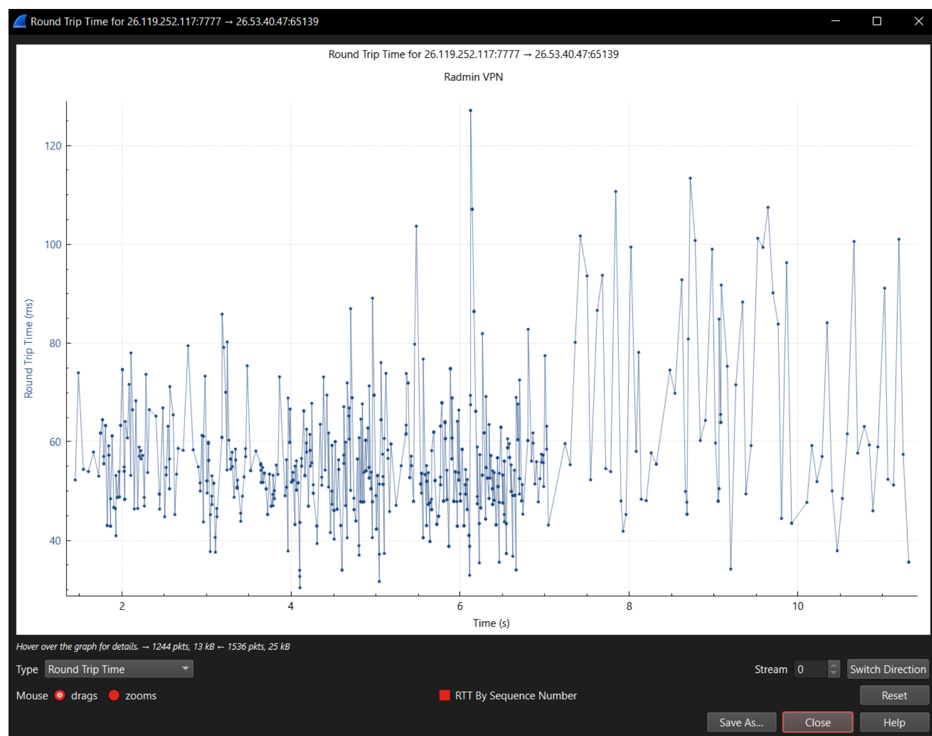

Рисунок 4.8 – Середня затримка при пропускній здатності в 75 МБіт/с

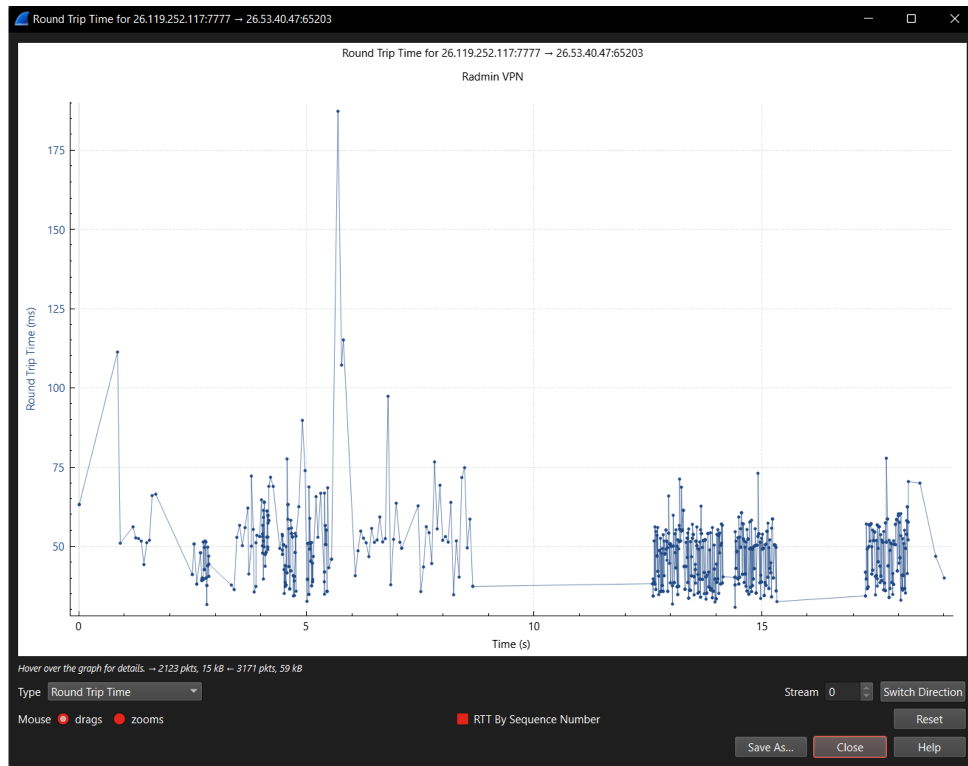

Рисунок 4.9 – Середня затримка при пропускній здатності в 18 МБіт/с

4.4 Аналіз ефективності системи

Комплексний аналіз ефективності розробленої мультиплеєрної системи базується на компонування результатів, отриманих під час тестування продуктивності при різній кількості підключених клієнтів та варіюванні мережевого навантаження. Інтеграція даних з попередніх експериментів дозволяє сформувати цілісне уявлення про функціональні можливості системи та визначити межі її ефективного застосування в реальних умовах експлуатації.

Аналіз залежностей між кількістю одночасно підключених користувачів та швидкістю мережевого з'єднання виявив складну нелінійну взаємодію цих параметрів. За оптимальних мережевих умов (швидкість понад 100 Мбіт/с) система демонструвала стабільну продуктивність навіть при підключенні до 15 клієнтів, зберігаючи затримку на рівні 50-65 мілісекунд. Проте при зниженні пропускної здатності до 15 Мбіт/с критична кількість користувачів зменшувалася до 5-10.

При одночасному збільшенні кількості клієнтів та зниженні швидкості мережі спостерігався ефект каскадної деградації продуктивності, коли незначне погіршення одного параметру призводило до диспропорційного зниження загальної ефективності системи.

Дослідження ефективності розподілу обчислювального навантаження показало, що серверна частина системи може опрацювати до 15+ одночасних з'єднань в оптимальних мережевих умовах. При цьому завантаження процесора складає 45-60%, а використання оперативної пам'яті не перевищує 10-15% від доступного обсягу. Такий розподіл ресурсів забезпечує необхідний резерв для обробки пікових навантажень та непередбачуваних ситуацій.

Результати комплексного аналізу підтверджують ефективність архітектурних рішень та алгоритмічних підходів, застосованих при розробці системи, і демонструють її готовність до практичного впровадження в реальних мультиплеєрних додатках середньої складності.

ВИСНОВКИ

У ході кваліфікаційної роботи було проведено аналіз сучасних технологій та готових рішень у галузі розробки мережевої архітектури для ігрових застосунків. На основі проведених досліджень, було виявлено основні проблеми сучасних моделей для ігрового рушія Unity. Основними проблемами виявилися невелика швидкість передачі даних, гнучкість та масштабованість у існуючих рішеннях.

Для вирішення виявленої проблеми з недостатньою швидкістю передачі даних було розроблено та реалізовано власну мережеву модель, що базується на сучасних підходах до організації мережевої взаємодії. Розроблена модель повністю відповідає сучасним стандартам проектування програмного забезпечення і з самого початку планувалася як масштабована система, здатна до подальших змін та оновлень без порушення цілісності архітектури.

Для вирішення основної проблеми зі швидкістю, були розроблені різні модулі відповідальності, які функціонують автономно та динамічно формують пакети даних оптимального розміру відповідно до встановлених пріоритетів. Ця модульна архітектура забезпечує ефективне використання мережевих ресурсів та мінімізує накладні витрати на передачу службової інформації.

Проведені експериментальні дослідження та комплексне тестування розробленої системи мережевої взаємодії для мультиплеєрних застосунків дозволили отримати важливі результати, що підтверджують ефективність обраних технологічних рішень та архітектурних підходів. Завдяки впровадженим інноваційним рішенням у результатах експериментального тестування можна спостерігати значне прискорення роботи системи, що наочно підтверджує ефективність розробленої архітектури.

Практична цінність отриманих результатів дослідження полягає у

можливості їх широкого застосування як у ігрових додатках, так і в суміжних системах, що потребують ефективної мережевої взаємодії. Більшість розроблених рішень, включаючи як програмну реалізацію, так і теоретичні основи, можуть бути адаптовані для використання в різних предметних областях при внесенні мінімальних додаткових оновлень та модифікацій.

Архітектурна гнучкість розробленої системи забезпечує розробникам можливість налаштування її параметрів відповідно до специфічних вимог конкретного проекту. Модульна структура системи дозволяє інтегрувати власний функціонал, впроваджувати нові протоколи передачі даних або застосовувати альтернативні алгоритми обробки мережевого трафіку без необхідності кардинальної перебудови всієї архітектури.

Перспективи подальшого розвитку розробленої системи відкривають широкі можливості для вдосконалення та розширення функціональності. Подальша розробка може включати інтеграцію підтримки додаткових мережевих протоколів або їх гібридних версій, що дозволить системі адаптуватися до специфічних вимог різних типів ігрових застосунків. Особливо перспективним напрямком є розробка адаптивних протоколів, здатних динамічно переключатися між різними режимами передачі даних залежно від поточних мережевих умов та характеристик ігрового контенту.

Загалом розроблена система є сформованим рішенням та гнучким каркасом для подальшої розробки інструментарію мережевої розробки, який дозволяє інтегрувати мережеву частину для ігрових застосунків. Надаючи розробникам відкриту платформу, що дає повну свободу для інтегрування або налаштування під поточні виклики та проблеми.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Alshammari A., Whittinghill D. Multiplayer Kinect Serious Games. International Journal of Game-Based Learning. 2015. Vol. 5, No 3. P. 45–61. URL: https://doi.org/10.4018/ijgbl.2015070104 (дата звернення: 02.06.2025).

2. An Introduction to Online Video Game QoS and QoE Influencing Factors / F. Metzger et al. IEEE Communications Surveys & Tutorials. 2022. P. 1. URL: https://doi.org/10.1109/comst.2022.3177251 (дата звернення: 02.06.2025).

3. Bryant B., Saiedian H. An evaluation of videogame network architecture performance and security. Computer networks. 2020. P. 17. URL: https://doi.org/10.1016/j.comnet.2021.03.015 (дата звернення: 02.04.2025).

4. Bura J. Making Multiplayer Games. Pro Android Web Game Apps. Berkeley, CA, 2012. P. 477–511. URL: https://doi.org/10.1007/978-1-4302-3820-1_12 (дата звернення: 02.06.2025).

5. Chen K.-T., Huang P., Lei C.-L. How sensitive are online gamers to network quality? Communications of the ACM. 2006. Vol. 49, No 11. P. 34–38. URL: https://doi.org/10.1145/1167838.1167859 (дата звернення: 02.06.2025).

6. Claypool M., Claypool K. Latency and player actions in online games. Communications of the ACM. 2006. Vol. 49, No 11. P. 40–45. URL: https://doi.org/10.1145/1167838.1167860 (дата звернення: 02.06.2025).

7. Frohnmayer M., Gift T. The TRIBES engine networking model. URL: https://www.gamedevs.org/uploads/tribes-networking-model.pdf (дата звернення: 02.06.2025).

8. Jessie D. T., Saari D. G. Multiplayer Games. Static & Dynamic Game Theory: Foundations & Applications. Cham, 2019. P. 153–174. URL: https://doi.org/10.1007/978-3-030-35847-1_6 (дата звернення: 02.06.2025).

9. Madhav S., Glazer J. Multiplayer Game Programming: Architecting Networked Games. Addison-Wesley Longman, Incorporated, 2015. 384 p.

10. Mirror Networking | Mirror. URL: https://mirror-

networking.gitbook.io/docs (дата звернення: 02.06.2025).

11. Pierre V. Beginner's Guide to Game Networking. URL: https://pvigier.github.io/2019/09/08/beginner-guide-game-networking.html (дата звернення: 02.06.2025).

12. Pun 2 - Introduction | Photon Engine. URL: https://doc.photonengine.com/pun/current/getting-started/pun-intro (дата звернення: 02.06.2025).

13. Quake Source Code Review. URL: https://fabiensanglard.net/quakeSource/quakeSourceNetWork.php (дата звернення: 02.06.2025).

14. Source Multiplayer Networking - Valve Developer Community. URL: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking (дата звернення: 02.06.2025).

15. The design of multiplayer online video game systems / C.-c. A. Hsu et al. ITCom 2003, Orlando, FL / ed. by A. G. Tescher et al. 2003. URL: https://doi.org/10.1117/12.512201 (дата звернення: 02.06.2025).

16. UNet Deprecation FAQ. URL: https://support.unity.com/hc/en-us/articles/360001252086-UNet-Deprecation-FAQ (дата звернення: 02.06.2025).

17. Yahyavi A., Kemme B. Peer-to-peer architectures for massively multiplayer online games. ACM Computing Surveys. 2013. Vol. 46, No 1. P. 1–51. URL: https://doi.org/10.1145/2522968.2522977 (дата звернення: 02.06.2025).

18. Сергеєв Д. В., Долгополов О. М. Застосування RLE алгоритму для створення ігрового контенту. 27-й Міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті» : міжнар. молодіж. форум, м. Харків, 2023 р. Харків, 2023. С. 129–130.

19. Сергеєв Д. В., Долгополов О. М. Особливості генерації структур 2D ігр на платформі Unity. 27-й Міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті» : міжнар. молодіж. форум, м. Харків, 2023 р. Харків, 2023. С. 131–132.

20. Фесенко Т.Г. Долгополов О.М., Сергеєв Д.В., Сергородцев І.Д.,

Жук М.В. Інформаційні технології для створення музично-ігрових проектів: бібліометричний аналіз. Збірник наукових праць. Системи управління, навігації та зв’язку, 2025, Том 2, №80.