МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«ПРОГРАМНА РЕАЛІЗАЦІЯ МЕХАНІЗМІВ КОМУНІКАЦІЇ ІГРОВИХ ЕЛЕМЕНТІВ МЕРЕЖЕВОЇ КОМП'ЮТЕРНОЇ ГРИ З АРХІТЕКТУРОЮ КЛІЄНТ-СЕРВЕР»**

Виконав: студент 4 курсу, групи 6.1219-2пі
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
К.П. Чорний
(ініціали та прізвище)
Керівник доцент кафедри програмної інженерії, доцент, к.ф.-м.н. Горбенко В.І.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент завідувач кафедри фундаментальної та прикладної математики, професор, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ 07 ” 02 2023 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Чорному Кирилу Павловичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Програмна реалізація механізмів комунікації ігрових елементів мережевої комп’ютерної гри з архітектурою клієнт-сервер

керівник роботи Горбенко Віталій Іванович, к.ф.-м.н, доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 26 » січня 2023 року № 102-с

2. Строк подання студентом роботи 07.06.2023 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Розробка бібліотеки, яка слугує як API для мережевого стеку комп’ютерних ігор.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою докладу

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 07.02.2023 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	08.02.2023	
2.	Збір вихідних даних.	20.02.2023	
3.	Обробка методичних та теоретичних	13.03.2023	
	джерел.		
4.	Розробка першого та другого розділу.	24.04.2023	
5.	Розробка третього розділу.	18.05.2023	
6.	Оформлення та нормоконтроль	01.06.2023	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	23.06.2023	

Студент _____ К.П. Чорний
(підпис) (ініціали та прізвище)

Керівник роботи _____ В.І. Горбенко
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Програмна реалізація механізмів комунікації ігрових елементів мережевої комп’ютерної гри з архітектурою клієнт-сервер»: 53 с., 12 рис., 8 джерел, 1 додаток.

АРХІТЕКТУРА КЛІЄНТ-СЕРВЕР, КОНСОЛЬНИЙ ІНТЕРФЕЙС, МЕРЕЖЕВА ГРА, ПАКУВАННЯ ДАНИХ, СИМЕТРИЧНЕ ШИФРУВАННЯ, JAVA.

Об’єкт дослідження – комп’ютерна мережева гра, способи пакування ігрових даних, способи передачі ігрових даних, архітектура мережевого модуля мережевої гри.

Мета роботи: дослідження переваг та недоліків мережевих архітектур. Дослідження переваг та недоліків протоколів передачі даних. Дослідження переваг та недоліків способів пакування ігрових даних. Розробка бібліотеки на мові програмування Java яка слугуватиме мережевим модулем в мережевих іграх.

Метод дослідження – аналітичний.

SUMMARY

Bachelor’s qualifying paper «Software Implementation of Game Elements Communication Mechanisms of Network Computer Game with Client-Server Architecture»: 53 pages, 12 figures, 8 references, 1 supplement.

CLIENT-SERVER ARCHITECTURE, CONSOLE INTERFACE, NETWORK GAME, DATA PACKAGING, SUMETRICAL ENCRYPTION, JAVA.

Object of the study is computer network game, ways of packaging game data, ways of data transferring, network game’s network module architecture

Aim of the study is pros and cons of network architectures. Pros and cons of data transferring protocols. Pros and cons of game data packaging ways. Developing a Java library which will be a network module for modern network games.

Method of research is analytical.

ЗМІСТ

ВСТУП

Сучасні комп’ютерні ігри діляться на 2 типи за кількістю гравців що можуть одночасно грати в гру: ігри створені для одного гравця, тобто singleplayer ігри та ігри в які може одночасно грати 2 та більше гравців, тобто multiplayer мережеві ігри. Обидва типи ігор можуть мати будь-який жанр, тип та платформу для якої створювалися, проте це завжди або гра для одного гравця або гра для декількох гравців. Відповідно до того скільки гравців може одночасно грати в гру, розробнику комп’ютерної гри слід відразу думати про те як краще за все розробити архітектуру гри.

Якщо це гра для одного гравця то розробнику легше робити архітектуру оскільки він не має перейматися про мережеву частину. Проте якщо в гру одночасно можуть грати 2 та більше гравців то розробнику слід думати про те як гравці будуть спілкуватися між собою та яким чином додати гравців в один ігровий світ.

Мета цієї роботи – це створити бібліотеку написану на Java яка полегшить розробку мережевої гри. Бібліотека візьме на себе усю мережеву частину від створення сервера, пакування даних які передаються від сервера до клієнта та навпаки, шифрування даних які передаються від сервера до клієнта та вбудований клієнтський фреймворк за допомогою якого можна полегшити під’єднання клієнта до сервера та передачу даних. Бібліотека має бути розроблена без сторонніх бібліотек включно та мати можливість поєднувати різні реалізації. В бібліотеці мають бути стандартні реалізації пакувальника, шифрувальника та вбудованого серверу з клієнтським фреймворком. Також в бібліотеці мають бути інтерфейси які розробник може сам реалізувати щоб поєднати стандартну реалізацію та свою реалізацію.

1 АНАЛІЗ МЕРЕЖЕВОЇ АРХІТЕКТУРИ СУЧАСНИХ МЕРЕЖЕВИХ ІГОР

1.1 Аналіз існуючих засобів передачі та пакування даних

Передача даних – це процес пересилання даних від одного комп’ютера до іншого/інших комп’ютера/комп’ютерів. На процес впливає мережева архітектура, протокол передачі та спосіб наповнення даними.

Розглянемо критерій основних мережевих архітектур: P2P та client-server.

P2P – (peer to peer) це розподілена архітектура мережі в якій кожен комп’ютер є незалежною частиною і називається вузол. Усі вузли рівні в правах та можуть виконувати будь-яку роботу яку може виконати інший вузол. Детально архітектуру P2P описано в роботі [1].

В архітектурі клієнт-сервер, клієнт є ініціатором спілкування з сервером, і, як правило, надає йому всю інформацію про себе [2]. Усі дані, які пересилаються від одного комп’ютера до іншого мають пройти через сервер.

Розглянемо критерій протоколу передачі: UDP та TCP.

UDP – (User Datagram Protocol) протокол передачі даних в якому не створюються прямі зв’язки між комп’ютерами, а дані передаються у довільному порядку [3]. Протокол розроблено у 1980-ому році, залишається актуальним для певних задач. Він був створений для швидкої передачі даних коли підтвердження цілісності переданих даних не потрібно.

TCP – (Transmission Control Protocol) з’єднано-орієнтований протокол, що виділяє сервер та клієнт [4]. Перший документ про його розробку з’явився у 1981, і на сьогоднішній день останній документ, що оновлює опис TCP, дотується серпнем 2022. На відміну від UDP транспортний протокол TCP контролює кількість переданих та отриманих даних, і дозволяє повторне надсилання тієї частини даних, яку не було отримано. Саме через це TCP

протокол є популярним у додатках онлайн ігор, оскільки для них важливим є повний контроль пересилання даних між клієнтом та сервером.

Сучасні мережеві додатки часто використовують для передавання даних формати JSON [5] та XML [6]. Для створення загальної бібліотеки ігрових додатків важливо розглянути переваги та недоліки цих форматів.

JSON (JavaScript Object Notation) є стандартом відображення Javascript об’єктів у текстовому вигляді. Не дивлячись на те що це стандарт в основному для Javascript, ця технологія набула широкого вживу поза межами цієї мови. Існує значна кількість бібліотек, написаних на різних мовах програмування, які підтримують парсинг JSON документів у їх нативні структури. Проте, пакувати JSON у пакет є доволі нераціональним використанням байтів, які відведено у пакеті. JSON формат має багато особливостей, щодо пакування, та займає багато місця. Не зважаючи на те що з даними у JSON форматі легше працювати, їх передавання мережею є доволі громіздкою та потребує певного обчислювального навантаження як на сервері, так і на клієнті. У сучасних іграх затримка на передачу даних є критичною, тому даний формат не є оптимальним для мережевих ігор.

XML (Extensible Markup Language) є форматом відображення даних, який призначено для людей та для машин [6]. Формат винайдено в 1996 році, проте він досі залишається популярним при написанні конфігураційних файлів та передачі даних. Пакування даних у форматі XML, як і JSON, доволі нераціональне, тому що значна частина пакету використовується для опису форматування. Передавання даних у форматі XML також є доволі громіздким та потребує певного обчислювального навантаження серверу, тому для сучасних мережевих ігор цей формат не є оптимальним.

Існує швидкий та раціональний спосіб пакування даних: байтовий масив. Усі змінні пакету пакуються у такий масив. Спосіб не потребує витрат на форматування бо кожна змінна на своєму місці займає певну кількість байтів. Пакування/розпакування відбувається швидше, оскільки дані складаються із змінних визначеного розміру.

1.2 Аналіз мережевих технологій мережевих ігор

Мережеві технології мережевої гри повинні бути надійними. Комп’ютери, поєднані в одну мережу мають спілкуватися без проблем. Усі дані, що надсилаються між комп’ютерами повинні надходити швидко та в повному обсязі. Щоб визначити найкращі технології мережевої гри, порівняємо їх.

Мережева архітектура відповідає за з’єднання комп’ютерів в одну мережу для пересилання ігрових даних між ними.

При використанні архітектури P2P всі комп’ютери рівні у правах. Кожен комп’ютер бачить усі інші комп’ютери під’єднані до цієї мережі. Так кожен з них може надіслати довільну кількість даних довільному комп’ютеру цієї мережі. В цій архітектурі невідомо який комп’ютер є головним, тому не відомо яка копія ігрового світу є справжньою. Комп’ютери в мережі можуть бути підключеними з різних країн світу. Якщо комп’ютеру треба передати дані з одного кінця світу в інший, затримка передачі може бути величезною. Комп’ютери в цій архітектурі вразливі до вірусів. Якщо один комп’ютер був зараженим, він може заразити інші комп’ютери в мережі. Комп’ютери в такій мережі будуть приймати/передавати багато даних, тому мають бути потужними щоб витримати таке навантаження.

У клієнт-серверній архітектурі відділяється сервер та клієнт. Сервер – головний комп’ютер через якого проходять усі дані. Клієнт – комп’ютер який доєднується до серверу для передачі ігрових даних. Сервер володіє інформацією про ігровий світ та про всіх клієнтів які були до нього підключені. Коли клієнт підключається до серверу, йому надсилається локальна копія світу серверу. Коли клієнт з чимось взаємодіє, пакет даних відправляється до серверу та надсилається до всіх клієнтів. Так клієнти залишаються синхронізованими з сервером. В клієнт-серверній архітектурі тільки сервер має бути потужним, бо через нього проходить весь трафік від клієнтів. Клієнти не приймають/надсилають дані до інших клієнтів, тому і не

потребують величезних обчислювальних можливостей. Ігрові сервери можуть знаходитися по всьому світу. Клієнти будуть доєднуватися до серверів ближче до себе і час на доставку пакету даних буде мінімальним. Для мережевих ігор оптимальним варіантом буде клієнт-серверна архітектура. Ця архітектура є централізованою, має маленьке навантаження на клієнтів та її можна розгортати по всьому світу.

Протокол передачі відповідає за відправку та прийом ігрових даних між комп'ютерами.

Протокол UDP не створює зв'язків між комп'ютерами. Коли клієнт підключається до серверу, серверу про це не повідомляється автоматично. Клієнт має сам надіслати пакет даних в якому він повідомляє що підключився до серверу. UDP відповідає за передачу даних без перевірки чи надійшли дані до кінця. Якщо дані загубилися десь на шляху до комп'ютера, про це ніяк не буде повідомлено. З іншого боку, передача даних протоколом UDP відбувається швидше, бо кожен пакет не треба перевіряти на доставку до іншого клієнту. Якщо клієнт від'єднається від серверу, він не зможе про це повідомити автоматично. Клієнту треба надіслати пакет в якому буде повідомлено про його відключення. Існує варіація протоколу UDP який перейняв повідомлення про доставку пакету від TCP. Цей підвид протоколу так само використовує UDP, проте слідкує за кожним пакетом. У випадку якщо від кінцевого комп'ютера не надійшов пакет з підтвердженням що пакет дійшов, він буде надісланий знову.

Протокол TCP є з'єднано-орієнтованим протоколом. Цей протокол перевіряє чи надійшли дані до кінцевого комп'ютера, та у випадку якщо не надійшли, відправить їх ще раз. Скільки разів комп'ютер буде намагатися відправити пакет з даними до іншого комп'ютера налаштовується в параметрах протоколу. Операційна система може самостійно розірвати зв'язок якщо між двома комп'ютерами певний час не пересилалися дані. Під час підключення клієнта до серверу між клієнтом та сервером відбудеться рукостискання. Це стандартна процедура у протоколі TCP. Таким чином

клієнт підтверджує свою присутність . Після рукостискання клієнт та сервер готові почати передавати дані один одному. Коли клієнт відключається від серверу, серверу буде автоматично про це повідомлено. Між клієнтом та сервером знову пройде рукостискання, тільки в цьому випадку вже з флагами закінчення зв’язку. Для мережевих ігор протокол передачі даних має бути надійним. Дані мають завжди приходити до комп’ютерів, не бути зміненими та у повному обсязі. Тому для мережевих ігор оптимальним варіантом є протокол передачі даних TCP.

Спосіб пакування відповідає за спосіб яким ігрові дані пакуються для наступної відправки їх. Пакувати дані можна як за стандартами, так і нативно.

Стандарт пакування об’єктів JSON є доволі зручним способом пакування даних. Хоч це і стандарт пакування JavaScript об’єктів в їх текстовий вигляд, цей стандарт застосовується і в інших мовах програмування. Стандарт може запакувати будь-який об’єкт, включаючи вкладені об’єкти в їх текстовий вигляд для подальшого пересилання. В стандарту є правила за якими він буде форматувати текстовий вигляд об’єкту. Через це текстовий вигляд об’єкту буде займати більше місця між сам об’єкт.

Стандарт пакування об’єктів XML є доволі зручним способом пакування даних. Насамперед цей стандарт пакування було винайдено для конфігураційних файлів. Проте з часом веб-сервери почали також його використовувати на противагу JSON. XML як і JSON перетворює об’єкти мов програмування в їх текстовий вигляд та підтримує вкладені об’єкти. Стандарт XML має ще більше правил форматування текстової варіації ніж JSON, тому займає значно більше місця ніж JSON версія.

Нативний Java Serializable – це вбудований спосіб пакування об’єктів у мові програмуванні Java. Кожен об’єкт який треба запакувати має реалізувати інтерфейс Serializable. Стандарт підтримує вкладені об’єкти. Усі вкладені об’єкти які мають бути запаковані повинні реалізовувати інтерфейс Serializable. Пакування цим стандартом пакує усі дані в байтову відповідність. Проте окрім байтової відповідності полів даних, до них також додається певна

інформація про поля. Ця інформація невелика за розміром, проте складає певний відсоток запакованих даних. Процес розпакування даних відбувається у зворотньому порядку. Всі байти даних разом з інформацією про поля зчитуються, та за допомоги інформації про поля JVM може швидше перевести дані у відповідні типи даних. Цей стандарт кращий ніж пакування у JSON чи XML за розміром даних та за швидкістю, проте розмір запакованих даних можна зменшити ще більше.

Пакування байтовим масивом – це спосіб пакування в якому всі поля об’єкту запаковуються в масив байтів. Під час пакування об’єкту всі поля запаковуються у певному порядку. Якщо дані будуть розпаковуватися в іншому порядку, виникне невизначена поведінка (undefined behaviour). Кожне поле займає певну кількість байтів яку відомо заздалегідь. Так розпакування байтового масиву у відповідний об’єкт відбувається швидко. Масив байтів складається тільки з даних полів, тому розмір даних є максимально малим.

В сучасних мережевих іграх мережеві технології мають забезпечувати надійний зв’язок, надійність передачі/прийому даних та швидкість обробки даних. Клієнт-серверна архітектура мережі підходить найбільше, оскільки вона централізована, та може розгортатися декілька разів у різних країнах. Протокол передачі даних TCP підходить більше ніж UDP, бо він з’єднано-орієнтований та стежить за пересиланням пакетів даних. Спосіб пакування байтовим масивом підходить більше ніж усі інші способи бо пакет даних складається тільки з даних що пересилаються та час на парсинг такого пакету мінімальний.

1.3 Постановка задачі

Мережеві ігри потребують мережевий модуль. Мережевий модуль – це модуль, що відповідає за пакування ігрових пакетів даних, шифрування пакетів даних та відправку/прийом пакетів даних. Під час розробки гри

розробник може використати вже готову бібліотеку, або сам реалізувати класи мережевого модуля. Щоб розробники могли більше часу зосередитися на грі, а не на інфраструктурі, є потреба у розробці бібліотеки яка слугувала би мережевим модулем.

В бібліотеці не має бути обмежень на тип даних який може передаватися. Кожна з частин бібліотеки має бути незалежною одна від одної, щоб за потреби розробник мережевої гри міг вимкнути будь-яку з цих частин. Мережева архітектура бібліотеки має бути клієнт-сервер, бо така архітектура є централізованою та легша в управлінні. Протокол передачі даних має бути TCP, оскільки це з’єднано-орієнтований протокол з відстеженням передачі/прийому пакетів даних. Спосіб пакування даних має бути байтовий масив оскільки всі дані пакету займають тільки ігрові дані та швидкість розпакування даних є найвищою.

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ БІБЛІОТЕКИ

2.1 Побудова вимог до бібліотеки

Сучасна мережева бібліотека для мережевих ігор має бути розділеною на 3 модуля: пакувальник пакетів даних, шифрувальник/дешифрувальник пакетів даних, вбудований сервер разом з клієнтським фреймворком. Також в бібліотеці має бути присутня стандартна реалізація для кожного модуля під назвою dflt, щоб розробник міг одразу вибрати стандартну реалізацію, а не розробляти свої класи. Усі класи стандартної реалізації мають мати префікс Default- щоб вирізнятися поміж усіх інших класів. Кожен модуль має бути написаний з використанням інтерфейсів, щоб дати розробнику мережевої гри свободу вибору та поєднання різних реалізацій.

Усі модулі бібліотеки мають бути незалежними. Модуль шифрувальника не має залежати від модуля пакувальника. Модуль вбудованого серверу не має залежати від модуля шифрувальника. Модулі мають приймати дані в певному уніфікованому вигляді. Таким виглядом може бути байтовий масив. Він має певні переваги поміж інших способів пакування даних. По-перше розмір пакету даних буде набагато менше, оскільки дані що передаються в пакеті це сирі дані пакету, без додаткового форматування. По-друге дані які передаються виглядають як байтовий масив, та для зловмисника який їх перехопив вони будуть виглядати як нісенітниця. По-третє дані швидше розпакувати оскільки пакет складається тільки з байтів даних які відповідають внутрішній структурі пакету даних та розмір кожного об’єкту відомий завжди.

Розглянемо вимоги до модуля пакувальника. Модуль пакувальника відповідає за пакування даних які будуть надіслані з серверу до клієнта чи навпаки. В модулі мають бути присутні інтерфейси які визначають об’єкт що має бути запакований та інтерфейс який пакує дані певного типу. Також має

бути клас в який дані будуть запаковані. Після того як всі дані були запаковані в клас, в класі має бути метод toArray щоб перетворити всі запаковані дані в байтовий масив. В модулі пакувальника мають бути присутні стандартні реалізації пакування всіх примітивів які існують в мові програмування. Так розробник мережевої гри зможе більше часу зосередитися на своїй грі, а не на реалізації класів які знаходяться в бібліотеці. Усі пакети які запаковує пакувальник мають мати визначений розмір, тобто їх розмір не має змінюватися. Якщо в пакету даних розмір даних буде змінюватися, пакувальник може помилитися під час пакування/розпакування а це призведе до невизначеної поведінки(undefined behaviour).

Розглянемо вимоги до модуля шифрувальника. Модуль шифрувальника відповідає за шифрування/дешифрування даних. В модулі мають бути присутні інтерфейси для шифрування/дешифрування на стороні серверу, та на стороні клієнту. Інтерфейси мають бути розроблені під певний комбінований протокол шифрування/дешифрування такий як RSA/AES-CBC [7, 8]. Використання комбінованих протоколів шифрування зможе покращити надійність шифрування/дешифрування пакетів даних та зробити так щоб зловмисник не зміг прочитати дані під час перехоплення. До початку шифрування між клієнтом та сервером має пройти так зване рукостискання. Це означає що клієнт та сервер узгодили між собою ключі які вони будуть використовувати для шифрування/дешифрування даних. Далі має відбутися генерація першого вектору який буде використовуватися разом з ключами для симетричного шифрування даних. Після цього клієнт та сервер зможуть шифрувати/дешифрувати дані один для одного.

Розглянемо вимоги до модуля вбудованого серверу з клієнтським фреймворком. Модуль вбудованого серверу складається з інтерфейсів основного серверного об’єкту, серверного роутеру, серверної конфігурації, серверних клієнтів, серверного обробника пакетів та серверних команд.

Основний серверний об’єкт це об’єкт який і запускає сервер. Цей об’єкт має метод serve який запускає сервер та починає слухати вхідні підключення.

Коли клієнт хоче підключитися сервер має зареєструвати клієнта та почати слухати вхідні дані які надходять з клієнту. Об’єкт має обслуговувати клієнтів у паралелі, щоб не утворювався блокуючий код.

Серверний роутер це об’єкт який розподіляє пакети даних які надійшли від клієнту до відповідних обробників пакетів. Сервер приймає пакети даних від клієнтів та передає їх у серверний роутер. Роутер дивиться на ідентифікатор пакету, та розподіляє пакет даних у відповідний обробник пакетів. Серверний роутер має розподіляти пакети у паралелі для того щоб не утворювався блокуючий код.

Серверна конфігурація це об’єкт який містить всю конфігурацію яка потрібна серверу та слугує контекстом серверу. Як конфігуратор: на якому порту працює сервер, чи використовується шифрування, яка фабрика робить об’єкти шифрування для кожного серверного клієнту. Як контекст: всі під’єднані клієнти, об’єкти шифрування клієнтів, серверні команди.

Серверні клієнти це об’єкти які представляють клієнта який підключився до серверу. В об’єктах є логіка блокуючого читання, відправки даних на сервер, закриття з’єднання з сервером, унікальний клієнтський ідентифікатор. Унікальний клієнтський ідентифікатор створюється з хосту клієнту, дві крапки, порт клієнту.

Серверний обробник пакету це об’єкти які оброблюють пакети даних відповідною логікою. Кожен пакет має унікальний ідентифікатор який підходить тільки одному серверному обробнику пакетів.

Серверні команди це об’єкти що відповідають командам які адміністратор серверу вводить в консоль. Кожна команда має доступ до серверної конфігурації щоб мати змогу редагувати його за потреби.

Модуль клієнтського фреймворку складається з інтерфейсів основного клієнтського об’єкту, клієнтської конфігурації, клієнтського роутеру та клієнтських обробників пакетів.

Основний клієнтський об’єкт це об’єкт який під’єднує клієнта до серверу. Для того щоб знати хост та порт на який підключатися йому потрібна клієнтська конфігурація.

Клієнтська конфігурація це об’єкт який містить інформацію про хост та порт серверу, чи використовується шифрування, який об’єкт клієнтського шифрувальника, всі клієнтські обробники пакетів. Для відправки та читання даних з серверу в конфігурації містяться методи блокуючого читання та написання даних до серверу.

Клієнтський роутер це об’єкт який перенаправляє дані які прийшли від сервера до відповідного клієнтського обробника пакетів. Кожен пакет даних що надійшов має оброблюватися в паралелі щоб не створювати блокуючий код.

Клієнтський обробник пакетів це об’єкти які оброблюють пакети даних які надсилаються з серверу. Кожен пакет даних має унікальний ідентифікатор який відповідає обробнику пакетів. Клієнтський роутер перенаправляє дані які прийшли до одного з обробників пакетів.

2.2 Проєктування класів бібліотеки

На рисунках 2.1 – 2.3 зображено спроєктовані діаграми модулів пакувальника, шифрувальника, вбудованого серверу з клієнтською частиною. На рисунку 2.1 зображено діаграму модуля packer. В цьому модулі присутні інтерфейси об’єкту який має бути запакований та об’єкту який буде пакувати, клас в який будуть запаковані дані та підмодуль dflt в якому знаходиться стандартна реалізація інтерфейсів доступна одразу. На рисунку 2.2 зображена діаграма модуля encryptor. В цьому модулі знаходяться інтерфейси для шифрування даних на сервері та клієнті і підмодуль dflt в якому знаходиться стандартна реалізація цих шифрувальників. В модулі embeddedserver

знаходяться 3 підмодулі: client, server, dflt в яких знаходяться інтерфейси клієнтського фреймворку, вбудованого серверу та стандартної реалізації.

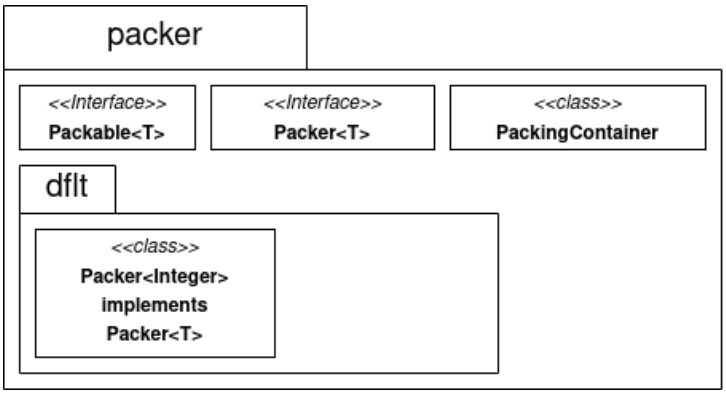

Рисунок 2.1 – Діаграма модуля пакувальника

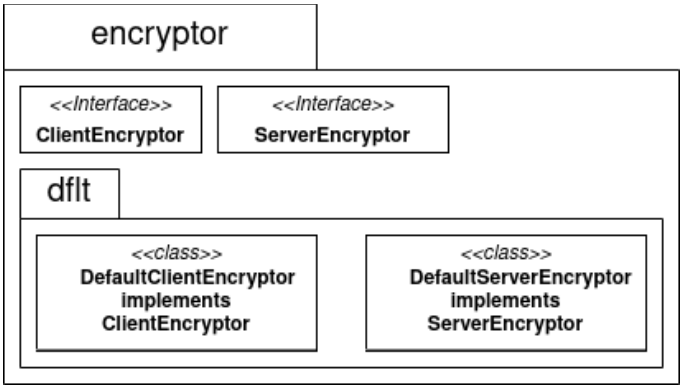

Рисунок 2.2 – Діаграма модуля шифрувальника

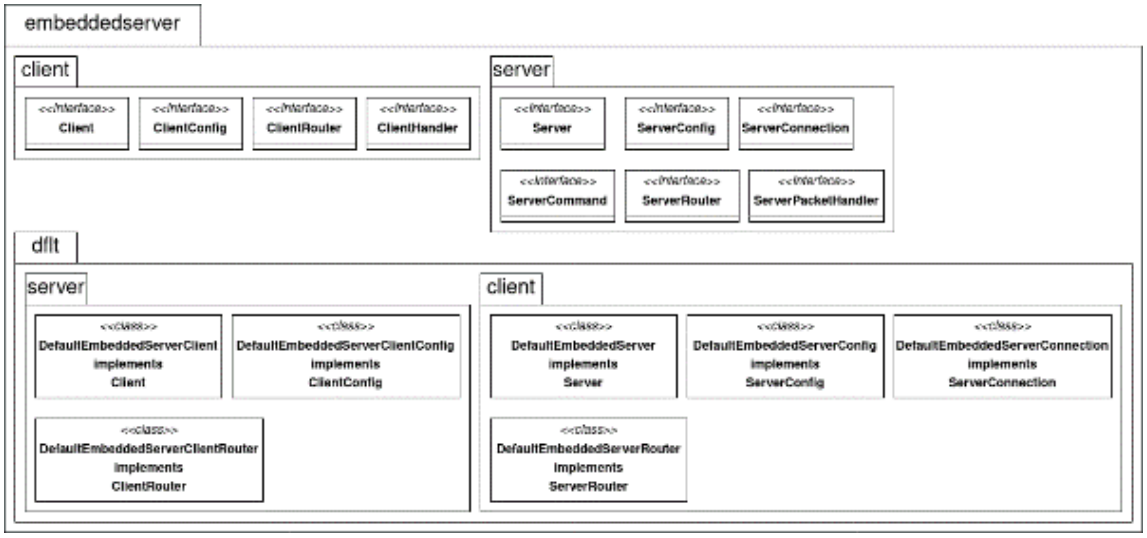

Рисунок 2.3 – Діаграма модуля вбудованого серверу

2.3 Реалізація класів бібліотеки

2.3.1 Реалізація пакувальника

Діаграму реалізації класів пакувальника показано на рис. 2.4.

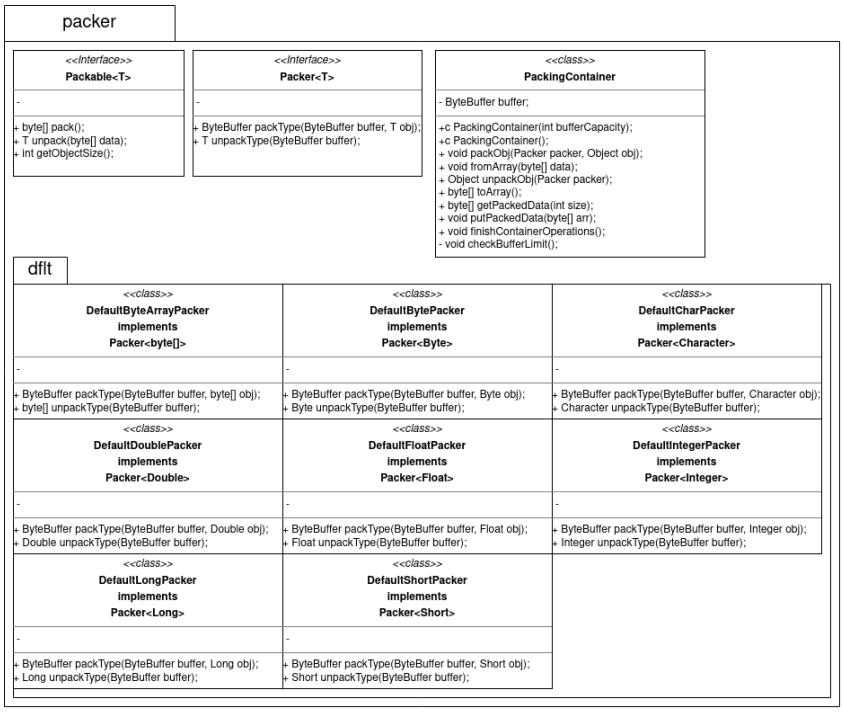

Рисунок 2.4 – Діаграма реалізації класів модуля packer

У модулі packer знаходяться 2 інтерфейси, 1 клас, а також підмодуль dflt, в якому є реалізації пакування всіх стандартних примітивів Java. Розглянемо принципи роботи класів модуля packer.

Інтерфейс Packable<T> відповідає за реалізацію пакування в об’єкті. Якщо певний клас має бути запакований в байтовий масив, то він має реалізувати цей інтерфейс. В цьому інтерфейсі знаходяться 2 основні методи для пакування та розпакування об’єкту відповідно: pack та unpack. Перший метод не приймає ніяких параметрів а лише запаковує клас в його байтову відповідність. Інший метод вже приймає байтовий масив який потім має перетворитися в об’єкт цього класу. Також існує метод який поверне загальний розмір об’єкту. Цей метод потрібен для того щоб розробник знав

скільки байтів йому треба передати в клас який буде пакувати щоб не було BufferOverflow чи BufferUnderflow виключень.

Інтерфейс Packer<T> відповідає за пакування/розпакування перного типу даних. Якщо певний тип даних використовується під час пакування об’єкту, то він має мати реалізацію його пакувальника(влаштована реалізація або написана самим розробником). В цьому класі знаходяться 2 методи які займаються пакуванням об’єкта типу та його розпакуванням відповідно. Для реалізації пакування чи розпакування об’єктів типів використовується вбудований в Java SDK інтерфейс ByteBuffer. Під час пакування цей інтерфейс перетвориться в клас HeapByteBuffer який розміщує всі байти які були в нього запаковані на Java купі. Метод packType відповідає за пакування об’єкта типу в ByteBuffer. У інтерфейсу ByteBuffer є влаштовані методи за допомогою яких можна запакувати всі примітиви Java. Метод unpackType відповідає за розпакування об’єкта типу з відповідного ByteBuffer. Розробник мережевої гри може використовувати стандартні влаштовані в бібліотеку вже написані реалізації пакування примітивів, а може використати свої написані.

Клас PackingContainer відповідає за зберігання запакованих в нього даних, а також пакування та розпакування з нього різних типів даних. Цей клас має 2 конструктори: перший конструктор використовується коли відома довжина даних(зазвичай під час пакування даних), а другий коли невідома довжина даних(зазвичай під час розпакування даних). В цьому класі присутній метод для пакування об’єктів певних типів даних в який треба передати пакувальник яким будуть пакувати та метод для розпакування об’єктів певного типу даних в який треба передати тип пакувальника яким він був запакований. Після того як всі потрібні дані були запаковані в контейнер, викликається метод toArray який поверне байтовий вигляд цих даних. Під час розпакування даних з їх байтового вигляду викликається метод fromArray який огорне внутрішній ByteBuffer в ті дані які були передані в нього. Після цього дані в ByteBuffer були встановлені, з цього контейнеру можна розпаковувати дані за допомогою unpackType. Якщо в даних є якась структура даних яка

вкладається в інші дані, клас підтримує вкладеність даних, проте тільки якщо відомий розмір цих даних (метод getObjectSize в класі Packable). Щоб запакувати вкладені дані треба викликати метод putPackedData. Методом getPackedData запаковані дані дістаються з контейнеру та передаються на розпакування. В класі присутній приватний метод checkBufferLimit який перевіряє ліміт контейнеру.

Окрім основних інструментів пакування/розпакування даних в бібліотеці також присутні стандартні реалізації пакування всіх примітивів Java (див. рис. 3.1). Це зроблено для того щоб розробник мережевої гри не витрачав час на реалізацію пакувальників, а одразу міг перейти до розробки мережевої гри. Усі вбудовані стандартні реалізації пакувальників для примітивів Java використовують методи в об’єкті ByteBuffer для пакування даних. Якщо це пакувальник типу int, тоді стандартний пакувальник такого ж типу викликає метод putInt об’єкту ByteBuffer. Усі стандартні реалізації мають префікс “Default-” який означає те що вони є вбудованими в бібліотеку. Проте є один стандартний пакувальник який відрізняється в реалізації від інших. Це DefaultByteArrayPacker. У цього пакувальника обмеження на пакування в ByteBuffer складає 1024 байти – 1022 байти даних та 2 байти, які вміщують розмір масиву байтів. Коли викликається метод packType в цьому пакувальнику то одразу перевіряється чи розмір масиву не перевищує розмір обмеження. Після цього всі дані які були передані в метод починають запаковуватися. Перші 2 байти використовуються щоб позначати розмір масиву. Після цього вираховується місце яке залишилося в остачі(якщо масив не був рівно 1022 байтів, він заповнюється порожніми даними). Заповнення порожніми даними зроблено для того щоб всі пакети мали фіксований розмір. Під час розпакування байтових даних спочатку визначається довжина масиву яка йде в перші 2 байти. Після цього створюється масив байтів з довжиною яку було визначено. Всі байти почергово дістаються з ByteBuffer та записуються в масив. Далі остаточні байти дістаються методом get щоб не руйнувати цілісність ByteBuffer.

2.3.2 Реалізація шифрувальника

Цей модуль відповідає за шифрування та дешифрування даних які передаються від сервера до клієнта та у зворотньому порядку. Діаграму реалізованих класів шифрувальника показано на рис. 2.5.

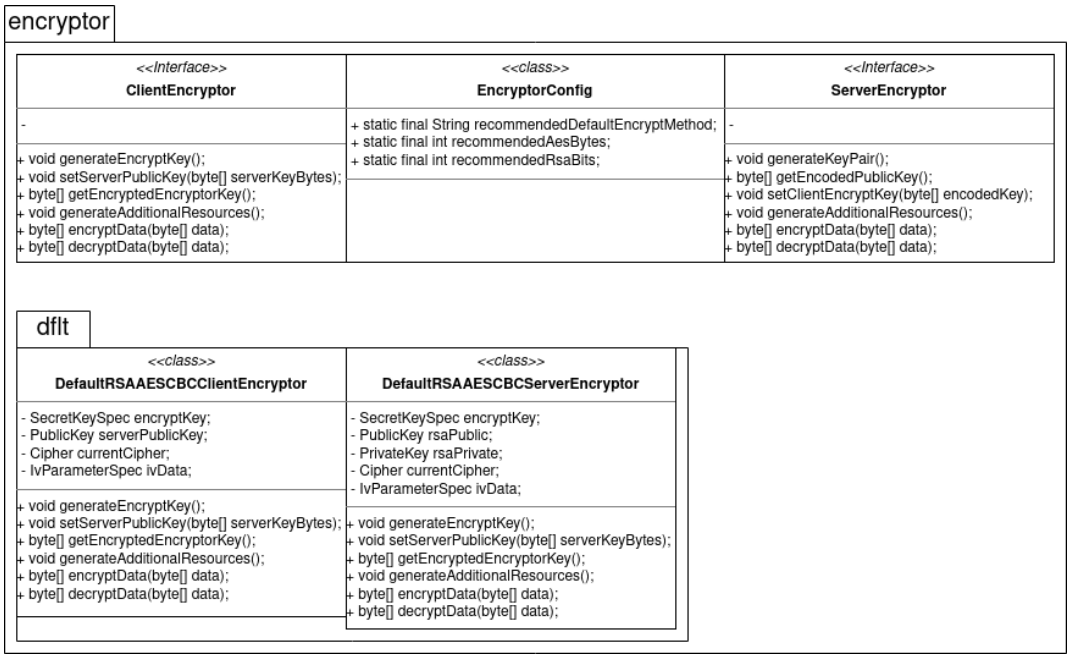

Рисунок 2.5 – Діаграма реалізованих класів модуля encryptor

У модулі encryptor знаходиться 2 інтерфейси шифрувальників(серверної та клієнтської частин) та клас який слугує конфігуратором шифрування. Шифрування закладене в ці інтерфейси це створення RSA/AES-CBC тунелю. Такий метод дозволяє безпечно передавати дані від одного комп’єютера до іншого. Для шифрування даних використовується симетричний метод AES-CBC. Для симетричного шифрування/дешифрування клієнт та сервер мають мати однаковий ключ. Клієнт має створити ключ та надіслати його серверу. Якщо він його надішле просто так, то зловмисник що буде сидіти поміж двома комп’ютерами перехопить цей ключ і шифрування не буде мати ніякого сенсу оскільки ключ було скомпрометовано. Для цього використовується захищений тунель RSA. Один комп’ютер генерує публічний та приватний ключ та відправляє публічний ключ до іншого комп’ютера. Інший шифрує свій

симетричний AES-CBC ключ та відправляє назад зашифрованим. Якщо зловмисник і перехопить симетричний ключ, нічого не зможе з ним зробити бо він вже зашифрований. Шлях створення захищеного з'єднання між двома комп'ютерами показано на рис 2.6.

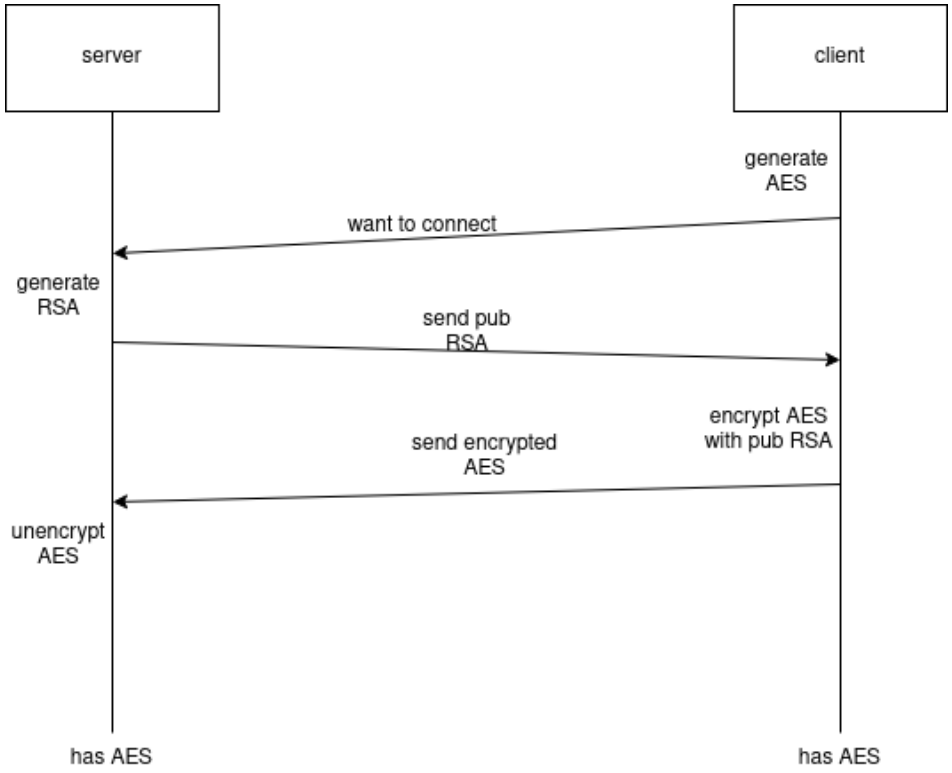

Рисунок 2.6 – Кроки створення RSA/AES-CBC

Після цих кроків між двома комп'ютерами пройшов обмін ключами та обидва мають симетричний ключ AES що буде використано для шифрування та дешифрування даних. Розглянемо принципи роботи інтерфейсів модуля encryptor.

ClientEncryptor відповідає за шифрування/дешифрування даних на стороні клієнта. Назви методів інтерфейсу йдуть в хронологічному порядку відповідно крокам створення безпечного RSA/AES-CBC з'єднання. Спочатку клієнт генерує симетричний AES-CBC ключ методом generateEncryptKey. Далі клієнт сповіщує сервер про готовність встановлення зв'язку. Сервер надсилає свій публічний RSA ключ. Клієнт викликає метод setServerPublicKey та передає в нього серверний ключ. Далі цим ключем шифрує свій AES ключ

методом getEncryptedEncryptionKey та надсилає серверу. Сервер дешифрує зашифрований AES ключ своїм приватним ключем. Симетричний AES ключ був переданий безпечно. Перед початком шифрування/дешифрування клієнт має викликати метод generateAdditionalResources один раз для генерації даних потрібних для шифрування. Клієнт може шифрувати дані методом encryptData. Дешифрувати методом decryptData.

ServerEncryptor відповідає за шифрування/дешифрування даних на стороні серверу. Назви методів інтерфейсу йдуть в хронологічному порядку відповідно крокам створення безпечного RSA/AES-CBC з'єднання. Спочатку клієнт повідомляє про підключення. Сервер генерує унікальну пару(публічний та приватний) ключів RSA методом generateKeyPair. Далі сервер перетворює публічний ключ методом getEncodedPublicKey та надсилає клієнту. Клієнт шифрує згенерований симетричний ключ та надсилає зашифрованим назад. Сервер встановлює зашифрований ключ методом setClientEncryptKey. Перед початком шифрування/дешифрування сервер має викликати метод generateAdditionalResources один раз для генерації даних потрібних для шифрування. Сервер може шифрувати дані методом encryptData та дешифрувати методом decryptData.

EncryptorConfig відповідає за конфігурацію шифрування/дешифрування. В класі знаходяться 3 публічні статичні поля: метод шифрування для стандартної реалізації, кількість байтів для генерації симетричного AES-CBC ключа та кількість бітів для генерації асиметричної пари RSA.

В підмодулі dflt знаходяться 2 розроблені класи шифрування/дешифрування для сервера та клієнта. Клас DefaultRSAAESCBCServerEncryptor для шифрування/дешифрування на стороні серверу та клас DefaultRSAAESCBCClientEncryptor для шифрування/дешифрування на стороні клієнту.

2.3.3 Реалізація вбудованого серверу

Цей модуль відповідає за створення вбудованого серверу та клієнтського фреймворку. Сам модуль складається з підмодулів client, server та dflt. У підмодулі server знаходяться класи для реалізації вбудованого серверу. Підмодуль client складається з класів для реалізації клієнтського фреймворку. Підмодуль dflt складається з підмодулів server та client в яких знаходяться вбудовані реалізації інтерфейсів модулів server та client. Діаграму реалізованих класів підмодуля сервера показано на рис. 2.7. Діаграму реалізованих класів підмодуля клієнта показано на рис. 2.8. Класи стандартної реалізації вбудованого серверу та клієнтського фреймворку мають схожу структуру. В обох підмодулях знаходяться основні об’єкти через яких підключаються, об’єкти конфігурації які також є контекстом та об’єкт роутеру який займається перенаправленням пакету даних до відповідного обробника пакетів. В обох підмодулях присутні інтерфейси обробників пакетів даних, проте не присутні стандартні реалізації оскільки це код ігри, а не інфраструктурний код.

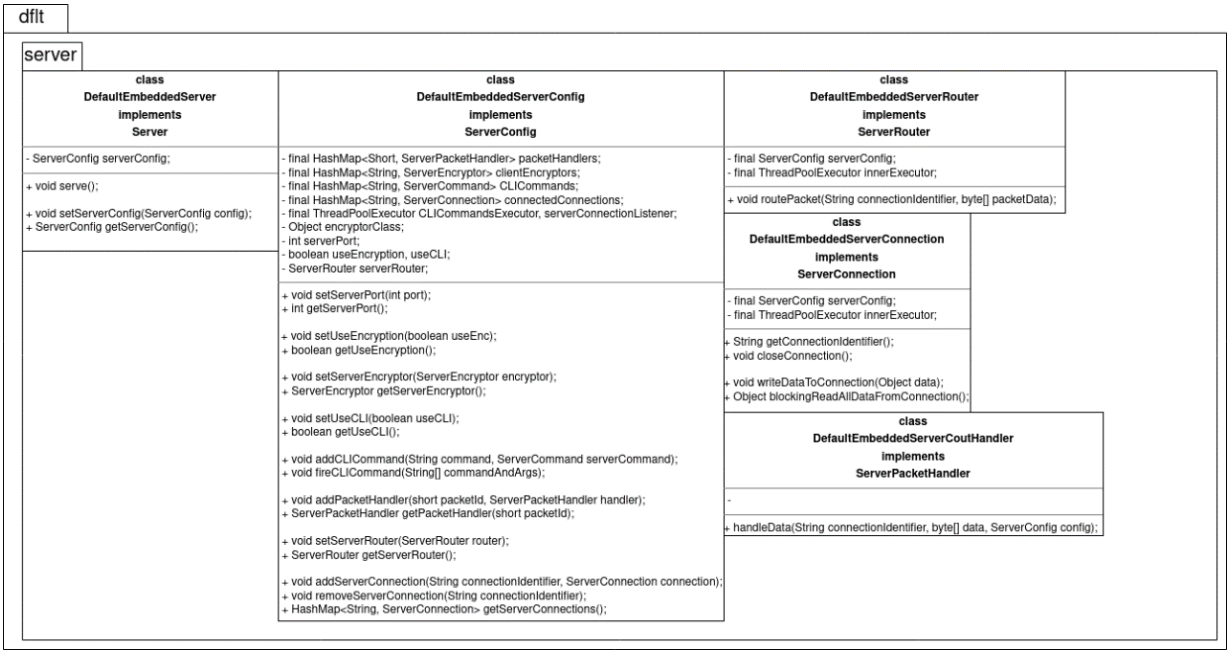

Рисунок 2.7 – Діаграма реалізованих класів підмодуля server

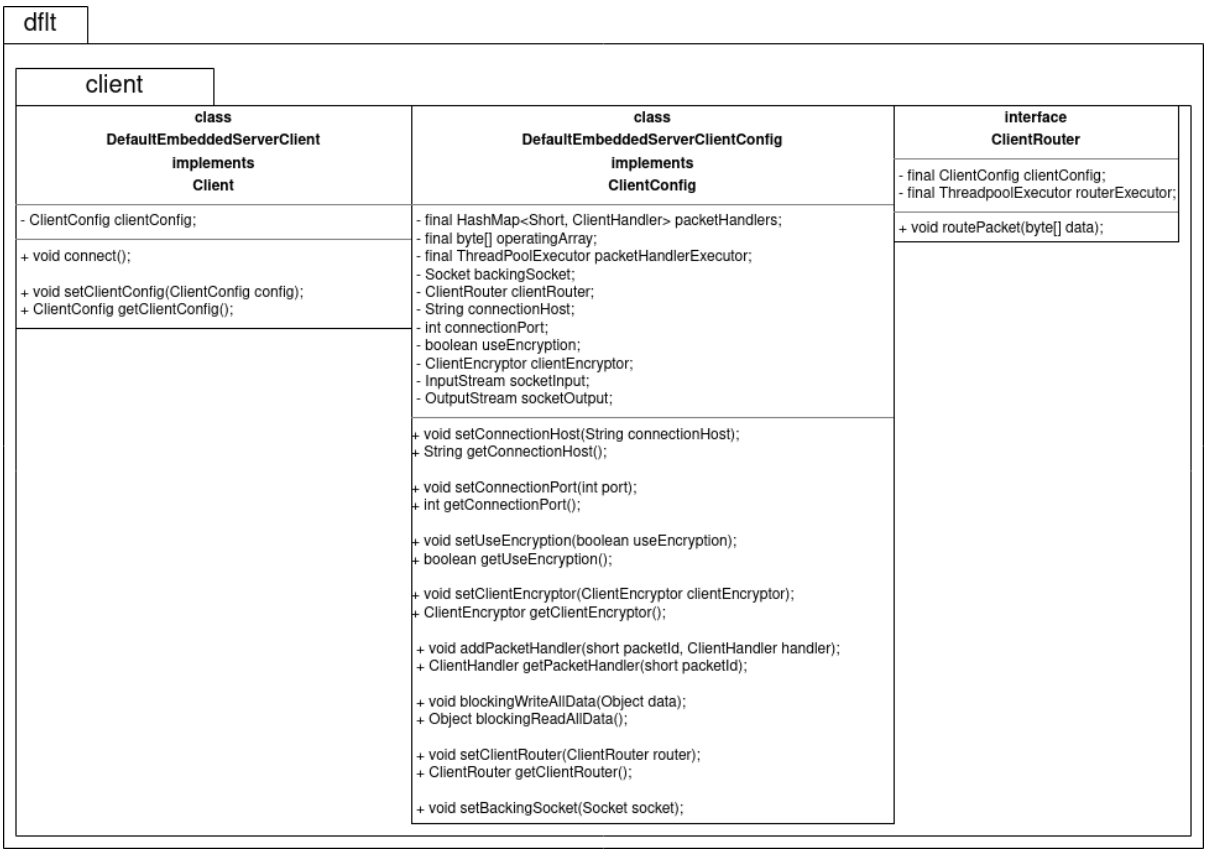

Рисунок 2.8 – Діаграма реалізованих класів підмодуля client

Розглянемо інтерфейси підмодуля server та їх реалізацію в підмодулі dflt.

Server відповідає за підключення клієнтів та прослуховування пакетів даних з них. Щоб клієнти могли підключитися до серверу використано ServerSocket вбудований в Java SDK. Об’єкт буде слухати нові підключення від клієнтів. Коли клієнт намагається підключитися, з метода accept буде повернуто Socket клієнта. Сервер створить об’єкт ServerConnectoin з Socket. Після чого додасть створений об’єкт у внітрішній ThreadPoolExecutor який опікується підключеними клієнтами. Він буде слухати дані які надходять з клієнта та передавати їх в ServerRouter. Також Socket буде додано у внутрішній HashMap серверної конфігурації. В ньому ключ це унікальний ідентифікатор клієнта(ім’я хоста + дві крапки + порт хоста). А створений об’єкт ServerConnection як значення. Якщо клієнт закрив з’єднання, метод який слухає вхідні дані з клієнта поверне –1. Сервер автоматично видалить Socket клієнта з внутрішнього HashMap та закриє з ним з’єднання. Стандартна

реалізація також містить обробник CLI команд. Розробник вводить команди у консолі запущеного серверу. Команду буде роздрібнено на командне слово(команда яку викликають) та аргументи команди. У конфігурації серверу буде знайдено відповідну команду яка відповідає за це командне слово та передано у внутрішній ThreadPoolExecutor який оброблює консольні команди.

ServerConfig відповідає за конфігурацію серверу та за внутрішній контекст серверу. Як конфігурація: порт на якому сервер буде запущено, чи використовується шифрування, якщо використовується з якого класу/фабрики створювати об’єкти шифрування, чи використовується командний інтерфейс(CLI). Як контекст: HashMap для зберігання підключених клієнтів, HashMap для зберігання об’єктів шифрування даних. В конфігурації є методи додавання обробників пакетів, консольних команд серверу, клієнтських підключень, встановлення об’єкту класу для створення об’єктів шифрування. В конфігурації присутні внутрішні ThreadPoolExecutor як для обробки вхідних даних клієнтів так і для обробки консольних команд. Кожен пакет даних чи консольна команда потрапляють на окремий потік свого ThreadPoolExecutor. ThreadPoolExecutor використаний для того щоб кожен пакет даних/консольна команда оброблювалися в паралелі та не створювали блокуючий код.

ServerConnection відповідає за огортання нативного Socket. Коли клієнт підключається до серверу, Socket буде повернуто з методу очікування нових підключень. Він буде огорнутий в ServerConnection, що має методи для взаємодії з клієнтом. В ServerConnection присутнє обмеження на розмір пакетів які можуть передаватися, для того щоб пакети не були надто довгими – 4 кілобайти (4096 байтів). Також присутні блокуючі методи для написання даних в клієнта та читання даних з клієнта. Ці методи блокуючі, оскільки невідомо коли клієнт буде взаємодіяти зі сервером.

ServerRouter відповідає за перенаправлення пакетів даних що надійшли до відповідного обробника пакетів. Кожен пакет оброблюється навколо try-statement. Якщо під час розбирання пакетів даних трапляється якась помилка, вона автоматично буде перехоплена та виведена як попередження в консоль.

ServerPacketHandler відповідає за обробку пакетів що надходять до сервера. В класі присутній метод в який передається ідентифікатор клієнту, байтовий масив даних який надійшов, та конфігурація серверу.

ServerCommand відповідає за команди які розробник може ввести через консольний інтерфейс. В інтерфейсі присутні 2 методи. Перший метод – встановлення серверної конфігурації. Другий – метод який займається логікою команди. В метод передається масив строк в який записане командне слово та всі аргументи які були передані. В вбудованій реалізації Server навколо обробника консольних команд стоїть try-statement. Якщо під час команди трапляється якась помилка, її буде виведено в консоль.

Розглянемо інтерфейси підмодуля client та їх реалізацію в підмодулі dflt.

Client інтерфейс який відповідає за підключення до серверу. В інтерфейсі присутні методи встановлення клієнтської конфігурації та підключення. Клієнт підключається до серверу за допомогою вбудованого Socket. Метод підключення не є блокуючим. Потік програми буде продовжувати далі йти в той час як мережевий потік буде працювати у фоні.

ClientConfig інтерфейс який відповідає за конфігурацію та контекст клієнту. В об’єкті знаходяться обробники пакетів, об’єкт шифрування/дешифрування, вхідні та вихідні потоки, роутер який буде перенаправляти пакети у відповідні обробники. В інтерфейсі присутні методи додавання обробників пакетів, методи блокуючого читання та запису на сервер.

ClientHandler інтерфейс який відповідає за обробку пакетів які надходять з серверу. В інтерфейсі присутній метод обробки вхідних даних в який передаються байтові дані пакету та клієнтська конфігурація.

ClientRouter інтерфейс який відповідає за передачу даних отриманих від серверу до відповідного обробника пакетів. В інтерфейсі присутній метод передачі байтових даних до відповідного обробника пакетів. В роутері присутній внутрішній ThreadPoolExecutor який займається передачею даних які надійшли від сервера до відповідного обробника пакетів.

Обидва підмодуля server та client мають стандартну реалізацію в підмодулі dflt. Стандартна реалізація розроблена з використанням стандартних, вбудованих в Java SDK засобів. Серверна реалізація для одночасної обробки багатьох клієнтів використовує ThreadPoolExecutor. Кожен під’єднаний клієнт автоматично додається в цей пул та з нього автоматично починають слухати дані. Коли дані з клієнта надходять, через серверний роутер вони передаються у відповідний обробник пакетів. У відповідному обробнику пакетів присутній свій пул потоків для того щоб не займати потік прослуховування клієнту. клієнтська реалізація схожа на серверну, тільки в клієнтській клієнт слухає тільки сервер. Коли з сервера надходять дані вони потрапляють у клієнтський роутер, а звідти вже у відповідний обробник пакетів. Усі пакети які надходять до клієнта чи сервера обробляються у паралелі щоб не існувало блокуючого коду.

3 ВИКОРИСТАННЯ БІБЛІОТЕКИ В КЛІЄНТСЬКИХ ТА СЕРВЕРНИХ ДОДАТКАХ

3.1 Приклад реалізації клієнта

Щоб перевірити стандартні реалізації клієнтського фреймворку вбудованого серверу розроблено простий клієнтський застосунок. Застосунок складається лише з одного класу, в якому написано всю логіку. Клієнт доєднається до серверу використовуючи дані передані в клієнтський конфігурації. Клієнт використовує стандартний клієнтський шифрувальник запропонований бібліотекою. Клієнт має обробники, що застосовуються для пакетів з ідентифікаторами 0 – 2.

Обробник пакету з ідентифікатором 0 додано для відлагодження. Цей обробник виводить в консоль дані які були в нього передані.

Обробник пакету з ідентифікатором 1 додано для етапу рукостискання в шифруванні. Сервер надсилає свій публічний ключ на цей обробник. Клієнт встановлює серверний ключ, шифрує свій ключ шифрування та надсилає назад зашифрований ключ.

Обробник пакету з ідентифікатором 2 додано для перевірки роботи шифрувальника. Дані які надходять на цей обробник є зашифрованими. Клієнт виводить на консоль зашифровані дані, дешифрує їх своїм шифрувальником та виводить на консоль дешифровані дані. У випадку якщо дані були дешифровані правильно на консолі клієнта та серверу будуть одинакові дані.

Код класу клієнту зображено в Додатку А.

3.2 Приклад реалізації сервера

Для того щоб перевірити стандартні реалізації серверної частини модуля вбудованого серверу розроблено простий серверний застосунок. Його складено з одного класу, в якому знаходиться вся логіка. Сервер запускається на порті, вказаному в серверній конфігурації. Використовується стандартна реалізація серверного шифрувальника для кожного клієнта. Серверу було додано обробники пакетів з ідентифікаторами 0 та 1.

Обробник пакету з ідентифікатором 0 доданий для відлагодження. Обробник виводить в консоль всі дані, які в нього надходять.

Обробник пакету з ідентифікатором 1 додано для етапу рукостискання з клієнтом. Клієнт надсилає свій зашифрований ключ до цього обробника. Сервер приймає зашифрований ключ, дешифрує його та встановлює його для подальшого шифрування/дешифрування.

До серверу також було додано 3 консольні команди: list, enc та msg.

Консольна команда list відповідає за відображення всіх під’єднаних клієнтів та їх шифрувальників. Команда не передбачає ніяких аргументів і у випадку, якщо вони будуть передані, їх буде проігноровано. Сервер візьме всіх під’єднаних клієнтів із серверної конфігурації та один за одним виведе їх у консоль.

Консольна команда enc відповідає за початок рукостискання з клієнтом. Команда передбачає, що перший аргумент це ім’я хосту клієнта, з яким буде проведено рукостискання. З конфігурації серверу буде отримано шифрувальник, створений для цього клієнта та надіслано публічний ключ клієнту, як початок рукостискання.

Консольна команда msg відповідає за відправку байтів із серверу до клієнта. Команда передбачає два або більше аргументів. Перший аргумент – це ім’я хосту клієнта, до якого будуть надіслані дані. Інші аргументи ТИРЕ – це дані, які буде переведено у байтовий вигляд для подальшої відправки. Далі сервер бере клієнтський шифрувальник та шифрує дані, які були введені через

консоль. Після цього буде виведено дані до та після шифрування і відправлено клієнту.

Код класу серверу зображено в Додатку А.

3.3 Тестування взаємодії сервера та клієнта

Для перевірки роботи клієнтського та серверного застосунків розроблено 3 класи що відповідають пакетам даних які будуть пересилатися. Пакет публічного ключа серверу, пакет зашифрованого ключа клієнта та пакет байтових даних. Кожен клас реалізує інтерфейс Packable<T> та використовує вбудовані в бібліотеку пакувальники для пакування даних. Коди класів зображено в Додатку 1. Обидва додатки збудовано та запаковано у jar архів разом з бібліотекою. Після цього запущено з консольного рядка без додаткових параметрів. Вивід серверу зображено на рисунку 3.1

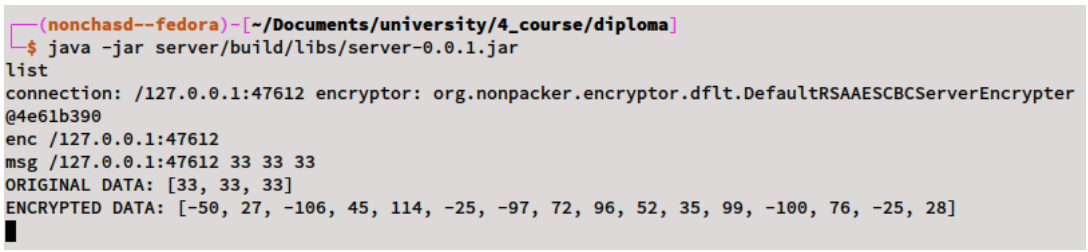

Рисунок 3.1 – Консольний вивід серверу

Після запуску серверного додатку, запущено клієнтський додаток. Командою list перевірено підключених клієнтів. Командою enc сервер почав проводити рукостискання з клієнтом. Командою msg виведено дані для пересилання та надіслано до клієнту. Вивід клієнта зображено на рисунку 3.2.

Після запуску серверу, був запущений клієнт. Запуск команди enc сервером змусив клієнта провести рукостискання. Коли на сервері була запущена команда msg, клієнт вивів шифровані та дешифровані дані що були йому надіслані.

```
┌──(nonchasd--fedora)-[~/Documents/university/4_course/diploma]
└─$ java -jar client/build/libs/client-0.0.1.jar
ORIGINAL DATA: [-50, 27, -106, 45, 114, -25, -97, 72, 96, 52, 35, 99, -100, 76, -25, 28]
DECRYPTED DATA: [33, 33, 33]
```

Рисунок 3.2 – Консольний вивід клієнта

На рисунку 3.3 зображено перехоплення пакетів між клієнтом та сервером. У випадку якщо зловмисник перехопить пакети даних вони для нього будуть нісенітницею.

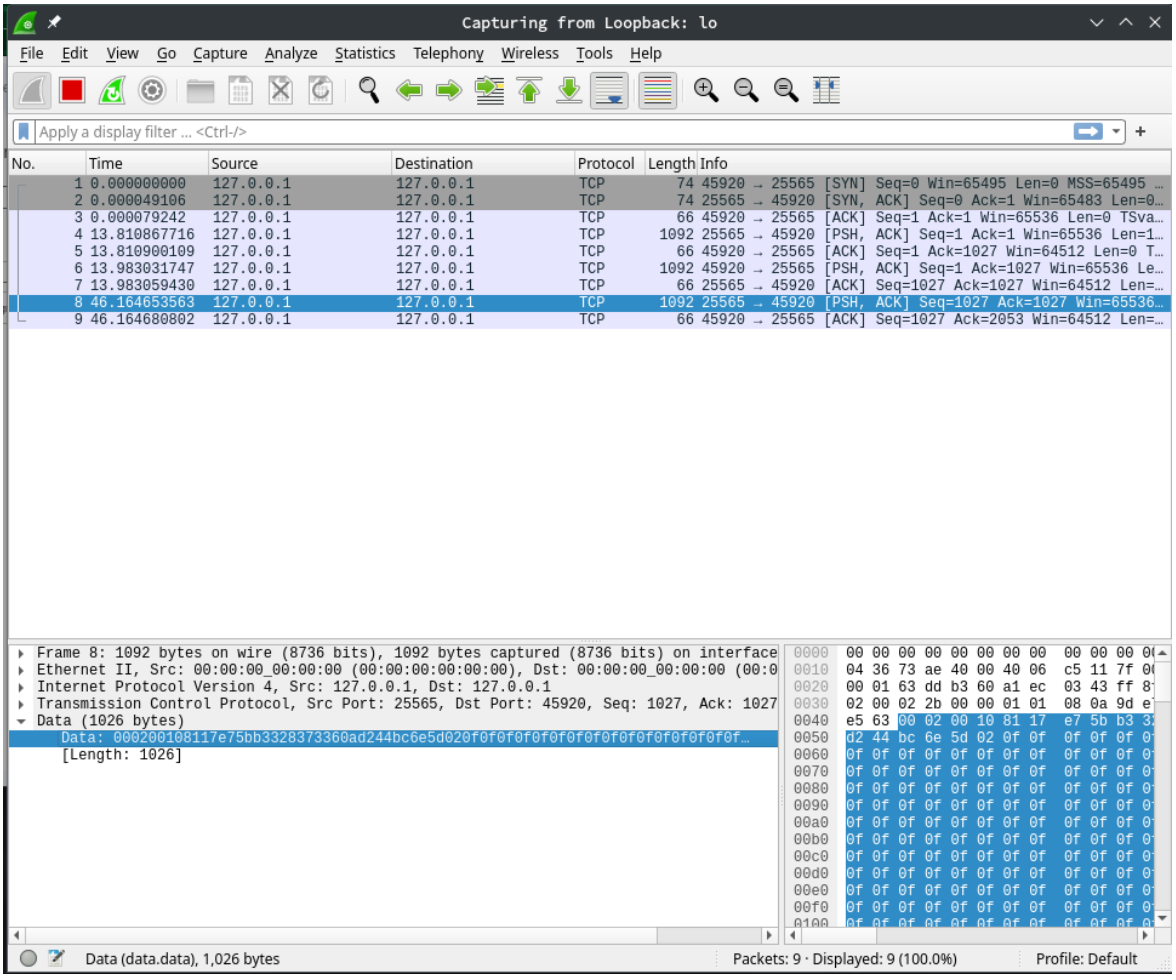

Рисунок 3.3 – Перехоплені пакети під час надсилання повідомлення

При вимкненому шифрувальнику дані які пересилаються легко прочитати. В перехопленому пакеті на рисунку 3.4 видно що пакет було надіслано до обробника пакетів з ідентифікатором 2, розмір байтового масиву 3 та дані які пересилаються.

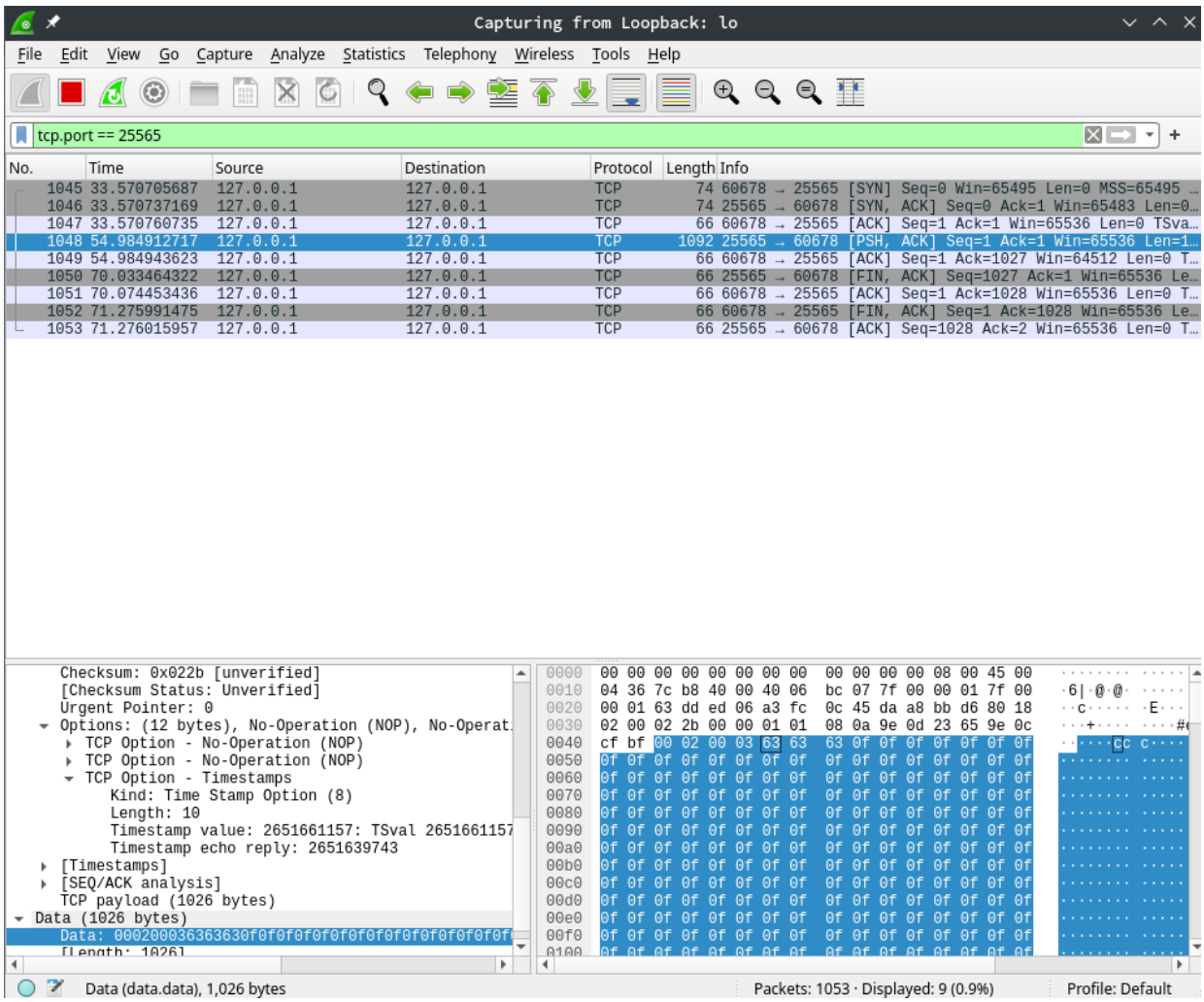

Рисунок 3.4 – Перехопений незашифрований пакет

ВИСНОВКИ

Сучасні мережеві ігри мають складну архітектуру. Окрім обробки користувацького вводу та рендерінгу гри ще присутня мережева складова. Мережева складова відповідає за підключення до серверу та спілкування з ним. Від клієнта серверу та навпаки надсилаються ігрові пакети даних. Це може бути як підключення до серверу, переміщення ігрового персонажу клієнтом чи зміна світу іншим гравцем.

Кожна зміна локального світу клієнта чи сервера – це явище. Коли явище відбувається на стороні клієнта, про це явище клієнт має одразу попередити сервер. Сервер отримає пакет з ігровими даними про це явище, обробить його та надішле оновлену копію світу всім іншим клієнтам. Якщо клієнт не надішле ігровий пакет явища, локальний світ клієнта та серверу буде різним. Так клієнт не зможе правильно орієнтуватися в ігровому світі та взаємодіяти з ним. Під час відправки ігрових даних їх треба шифрувати.

Якщо пакети ігрових даних не шифрувати, зловмисник може їх перехопити, розпакувати та нашкодити гравцю чи серверу. Найліпший спосіб шифрування – це створення захищеного RSA/AES-CBC тунелю. Цей тунель використовує симетричний AES-CBC алгоритм для шифрування. Проте сам ключ шифрування пересилається через RSA тунель. Таким чином в клієнта та сервера є ключ який був переданий безпечно без ризику бути перехопленим. Проте не завжди треба шифрувати ігрові пакети. Якщо всі комп’ютери клієнтів знаходяться в одній локальній мережі, шифрувати пакети даних не треба.

ПЕРЕЛІК ПОСИЛАНЬ

1. Camarillo G. Rfc p2p. URL: https://www.rfc-editor.org/rfc/pdfrfc/rfc5694.txt.pdf (дата звернення: 17.03.2023).

2. Contributors to Wikimedia projects. Client-server model – Wikipedia. Wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/wiki/Client-server_model (дата звернення: 16.03.2023).

3. Postel J. Rfc udp. URL: https://www.rfc-editor.org/rfc/rfc768 (дата звернення: 15.04.2023).

4. Eddy W. Rfc tcp. URL: https://www.rfc-editor.org/rfc/rfc9293.pdf (дата звернення: 24.03.2023).

5. Bray T. Rfc json. URL: https://www.rfc-editor.org/rfc/rfc8259 (дата звернення: 01.04.2023).

6. Hollenbeck S., Rose M., Masinter L. Rfc xml. URL: https://www.rfc-editor.org/rfc/rfc3470 (дата звернення: 23.03.2023).

7. Rfc rsa / K. Moriarty та ін. URL: https://www.rfc-editor.org/rfc/rfc8017 (дата звернення: 28.02.2023).

8. Frankel S., Glenn R., Kelly S. Rfc aes-cbc. URL: https://www.rfc-editor.org/rfc/rfc3602 (дата звернення: 13.04.2023).

ДОДАТОК А

Коди класів серверного та клієнтського додатків

А.1 Код класу публічного ключа серверу

```
package org.nonchasd.packets;

import org.nonpacker.packer.Packable;
import org.nonpacker.packer.PackingContainer;
import org.nonpacker.packer.dflt.DefaultByteArrayPacker;
import org.nonpacker.packer.dflt.DefaultShortPacker;

import java.util.Objects;

public class EncryptedPublicKeyPacket implements
Packable<EncryptedPublicKeyPacket> {

    public static final int encodedPublicKeyPacketSizeBytes =
DefaultByteArrayPacker.maxArraySize + Short.BYTES;
    private final DefaultByteArrayPacker arrayPacker;

    private final DefaultShortPacker packetIdPacker;
    private byte[] publicKeyData;
    private PackingContainer container;

    public EncryptedPublicKeyPacket() {
        this.arrayPacker = new DefaultByteArrayPacker();
        this.packetIdPacker = new DefaultShortPacker();
    }
```

```
    @Override
    public byte[] pack() {
        if (Objects.isNull(this.container)) this.container = new
PackingContainer(this.getObjectSize());

        this.container.packObj(this.packetIdPacker, (short) 1);

        this.container.packObj(this.arrayPacker, this.publicKeyData);

        this.container.finishContainerOperations();

        return this.container.toArray();
    }

    @Override
    public EncryptedPublicKeyPacket unpack(byte[] data) {
        if (Objects.isNull(this.container))
            this.container = new PackingContainer(0);

        this.container.fromArray(data);

        this.container.unpackObj(this.packetIdPacker);

        this.publicKeyData = (byte[]) this.container.unpackObj(this.arrayPacker);

        this.container.finishContainerOperations();

        return this;
    }
```

```
    @Override
    public int getObjectSize() {
        return EncryptedPublicKeyPacket.encodedPublicKeyPacketSizeBytes;
    }

    public byte[] getPublicKeyData() {
        return this.publicKeyData;
    }

    public void setPublicKeyData(byte[] publicKeyData) {
        this.publicKeyData = publicKeyData;
    }
}
```

А.2 Код класу зашифрованого ключа клієнту

```
package org.nonchasd.packets;

import org.nonpacker.packer.Packable;
import org.nonpacker.packer.PackingContainer;
import org.nonpacker.packer.dflt.DefaultByteArrayPacker;
import org.nonpacker.packer.dflt.DefaultShortPacker;

import java.util.Objects;

public class EncryptedEncryptKeyPacket implements
Packable<EncryptedEncryptKeyPacket> {
```

```
    public static final int encryptedEncryptKeyPacketSizeBytes =
DefaultByteArrayPacker.maxArraySize + Short.BYTES;
    private byte[] encryptedEncryptKey;

    private PackingContainer container;

    private final DefaultByteArrayPacker byteArrayPacker;

    private final DefaultShortPacker packetIdPacker;

    public EncryptedEncryptKeyPacket() {
        this.byteArrayPacker = new DefaultByteArrayPacker();
        this.packetIdPacker = new DefaultShortPacker();
    }



    @Override
    public byte[] pack() {
        if (Objects.isNull(this.container)) this.container = new
PackingContainer(this.getObjectSize());

        this.container.packObj(this.packetIdPacker, (short) 1);

        this.container.packObj(this.byteArrayPacker, this.encryptedEncryptKey);

        this.container.finishContainerOperations();

        return this.container.toArray();
    }
```

```
    @Override
    public EncryptedEncryptKeyPacket unpack(byte[] data) {
        if (Objects.isNull(this.container)) this.container = new PackingContainer();

        this.container.fromArray(data);

        this.container.unpackObj(this.packetIdPacker);

        this.encryptedEncryptKey = (byte[])
this.container.unpackObj(this.byteArrayPacker);

        this.container.finishContainerOperations();

        return this;
    }

    @Override
    public int getObjectSize() {
        return EncryptedEncryptKeyPacket.encryptedEncryptKeyPacketSizeBytes;
    }

    public byte[] getEncryptedEncryptKey() {
        return this.encryptedEncryptKey;
    }

    public void setEncryptedEncryptKey(byte[] encryptedEncryptKey) {
        this.encryptedEncryptKey = encryptedEncryptKey;
    }
}
```

А.3 Код класу байтових даних

```
package org.nonchasd.packets;

import org.nonpacker.packer.Packable;
import org.nonpacker.packer.PackingContainer;
import org.nonpacker.packer.dflt.DefaultByteArrayPacker;
import org.nonpacker.packer.dflt.DefaultShortPacker;
import java.util.Objects;

public class EncryptedSendEncryptedMessagePacket implements
Packable<EncryptedSendEncryptedMessagePacket> {

    private PackingContainer container;

    public static final int encryptedSendEncryptedMessagePacketSizeBytes =
DefaultByteArrayPacker.maxArraySize + Short.BYTES;

    private DefaultByteArrayPacker arrayPacker;

    private DefaultShortPacker shortPacker;

    private byte[] data;

    public EncryptedSendEncryptedMessagePacket(){
        this.arrayPacker = new DefaultByteArrayPacker();
        this.shortPacker = new DefaultShortPacker();
    }

    @Override
```

```
public byte[] pack() {
    if(Objects.isNull(this.container))
        this.container = new PackingContainer(this.getObjectSize());

    this.container.packObj(this.shortPacker, (short)2);

    this.container.packObj(this.arrayPacker, this.data);

    this.container.finishContainerOperations();

    return this.container.toArray();
}

@Override
public EncryptedSendEncryptedMessagePacket unpack(byte[] data) {
    if (Objects.isNull(this.container))
        this.container = new PackingContainer();

    this.container.fromArray(data);

    this.container.unpackObj(this.shortPacker);

    this.data = (byte[]) this.container.unpackObj(this.arrayPacker);

    this.container.finishContainerOperations();

    return this;
}

@Override
```

```
    public int getObjectSize() {
        return
EncryptedSendEncryptedMessagePacket.encryptedSendEncryptedMessagePacketS
izeBytes;
    }

    public byte[] getData() {
        return this.data;
    }

    public void setData(byte[] data) {
        this.data = data;
    }
}
```

A.4 Код серверного додатку

```
package org.server;

import org.nonchasd.packets.EncryptedEncryptKeyPacket;
import org.nonchasd.packets.EncryptedPublicKeyPacket;
import org.nonchasd.packets.EncryptedSendEncryptedMessagePacket;
import org.nonpacker.embeddedserver.dflt.server.DefaultEmbeddedServer;
import org.nonpacker.embeddedserver.dflt.server.DefaultEmbeddedServerConfig;
import org.nonpacker.embeddedserver.dflt.server.DefaultEmbeddedServerRouter;
import org.nonpacker.embeddedserver.server.ServerCommand;
import org.nonpacker.embeddedserver.server.ServerConfig;
import org.nonpacker.embeddedserver.server.ServerConnection;
import org.nonpacker.encryptor.ServerEncryptor;
import org.nonpacker.encryptor.dflt.DefaultRSAAESCBCServerEncrypter;
```

```
import java.util.Arrays;

public class Entry {
    public static void main(String[] args) {
        DefaultEmbeddedServerConfig serverConfig = new
DefaultEmbeddedServerConfig();
        DefaultEmbeddedServer server = new DefaultEmbeddedServer();
        DefaultEmbeddedServerRouter serverRouter = new
DefaultEmbeddedServerRouter(serverConfig);

        serverConfig.setServerRouter(serverRouter);

        serverConfig.setUseEncryption(true);

serverConfig.setServerSideEncryptorClass(DefaultRSAAESCBCServerEncrypter.
class);

        serverConfig.setServerPort(25565);

        serverConfig.setUseCLI(true);

        serverConfig.addPacketHandler((short) 0, (connectionIdentifier, data, config)
–> {
            System.out.println(Arrays.toString(data));
        });

        ServerCommand listAllConnectionCommand = new ServerCommand() {

            private ServerConfig config;
```

```
        @Override
        public void setServerConfig(ServerConfig config) {
            this.config = config;
        }

        @Override
        public void toTask(String[] commandAndArgs) {
            for (String connectionIdentifier :
this.config.DEBUG_getServerConnections().keySet()) {
                DefaultRSAAESCBCServerEncrypter connectionEncryptor =
(DefaultRSAAESCBCServerEncrypter)
this.config.getServerSideEncryptor(connectionIdentifier);

                System.out.println("connection: " + connectionIdentifier + "
encryptor: " + connectionEncryptor);
            }
        }
    };

    listAllConnectionCommand.setServerConfig(serverConfig);

    serverConfig.addCLICommand("list", listAllConnectionCommand);

    ServerCommand encryptConnectionCommand = new ServerCommand() {
        private ServerConfig config;

        @Override
        public void setServerConfig(ServerConfig config) {
            this.config = config;
```

```
        }

        @Override
        public void toTask(String[] commandAndArgs) {
            String host = commandAndArgs[1];

            ServerEncryptor hostEncryptor =
this.config.getServerSideEncryptor(host);
            ServerConnection hostConnection =
this.config.DEBUG_getServerConnections().get(host);

            EncryptedPublicKeyPacket packet = new EncryptedPublicKeyPacket();
            packet.setPublicKeyData(hostEncryptor.getEncodedPublicKey());

            byte[] packetData = packet.pack();

            hostConnection.DEBUG_blockingWriteDataToConnection(packetData);
        }
    };

    encryptConnectionCommand.setServerConfig(serverConfig);

    serverConfig.addCLICommand("enc", encryptConnectionCommand);

    serverConfig.addPacketHandler((short) 1, (connectionIdentifier, data, config)
–> {
        EncryptedEncryptKeyPacket incomingPacket = new
EncryptedEncryptKeyPacket();

        incomingPacket.unpack(data);
```

```
            ServerEncryptor hostEncryptor =
config.getServerSideEncryptor(connectionIdentifier);


hostEncryptor.setClientEncryptKey(incomingPacket.getEncryptedEncryptKey());

            hostEncryptor.generateAdditionalResources();
        });
        ServerCommand sendMsgCommand = new ServerCommand() {

            private ServerConfig config;

            @Override
            public void setServerConfig(ServerConfig config) {
                this.config = config;
            }

            @Override
            public void toTask(String[] commandAndArgs) {
                EncryptedSendEncryptedMessagePacket outMessage = new
EncryptedSendEncryptedMessagePacket();
                String host = commandAndArgs[1];

                ServerConnection hostConnection =
this.config.getServerConnection(host);
                ServerEncryptor hostEncryptor =
this.config.getServerSideEncryptor(host);

                String[] stringBytes = Arrays.copyOfRange(commandAndArgs, 2,
```

```
commandAndArgs.length);

            byte[] bytes = new byte[stringBytes.length];

            for (int idx = 0; idx < stringBytes.length; idx++)
               bytes[idx] = Byte.parseByte(stringBytes[idx]);

            System.out.println("ORIGINAL DATA: " + Arrays.toString(bytes));

            byte[] encryptedBytes = hostEncryptor.encryptData(bytes);

            System.out.println("ENCRYPTED DATA: " +
Arrays.toString(encryptedBytes));

            outMessage.setData(encryptedBytes);


hostConnection.DEBUG_blockingWriteDataToConnection(outMessage.pack());
         }
      };

      sendMsgCommand.setServerConfig(serverConfig);

      serverConfig.addCLICommand("msg", sendMsgCommand);

      server.setServerConfig(serverConfig);

      server.serve();
   }
}
```

А.5 Код клієнтського додатку

```
package org.client;

import org.nonchasd.packets.EncryptedEncryptKeyPacket;
import org.nonchasd.packets.EncryptedPublicKeyPacket;
import org.nonchasd.packets.EncryptedSendEncryptedMessagePacket;
import org.nonpacker.embeddedserver.dflt.client.DefaultEmbeddedServerClient;
import
org.nonpacker.embeddedserver.dflt.client.DefaultEmbeddedServerClientConfig;
import
org.nonpacker.embeddedserver.dflt.client.DefaultEmbeddedServerClientRouter;
import org.nonpacker.encryptor.dflt.DefaultRSAAESCBCClientEncrypter;

import java.util.Arrays;

public class Entry {

    public static void main(String[] args) throws Throwable {
        DefaultEmbeddedServerClient client = new DefaultEmbeddedServerClient();
        DefaultEmbeddedServerClientConfig clientConfig = new
DefaultEmbeddedServerClientConfig();
        DefaultEmbeddedServerClientRouter clientRouter = new
DefaultEmbeddedServerClientRouter(clientConfig);

        clientConfig.setConnectionHost("127.0.0.1");
        clientConfig.setUseEncryption(true);
        clientConfig.setClientEncryptor(new
DefaultRSAAESCBCClientEncrypter());
        clientConfig.getClientEncryptor().generateEncryptKey();
```

```
        clientConfig.setConnectionPort(25565);
        clientConfig.setClientRouter(clientRouter);

        client.setClientConfig(clientConfig);

        clientConfig.addPacketHandler((short) 0, (data, config) –> {
            System.out.println(Arrays.toString(data));
        });

        clientConfig.addPacketHandler((short) 1, (data, config) –> {
            EncryptedPublicKeyPacket encryptedPublicKeyPacket = new
EncryptedPublicKeyPacket();

            encryptedPublicKeyPacket.unpack(data);


config.getClientEncryptor().setServerPublicKey(encryptedPublicKeyPacket.getPu
blicKeyData());

            byte[] encryptedEncryptKey =
config.getClientEncryptor().getEncryptedEncryptKey();
            EncryptedEncryptKeyPacket packet = new EncryptedEncryptKeyPacket();

            packet.setEncryptedEncryptKey(encryptedEncryptKey);

            byte[] packetData = packet.pack();

            config.DEBUG_blockingWriteAllData(packetData);

            config.getClientEncryptor().generateAdditionalResources();
```

```
        });

        clientConfig.addPacketHandler((short) 2, (data, config) –> {
            EncryptedSendEncryptedMessagePacket incomingMessage = new
EncryptedSendEncryptedMessagePacket();

            incomingMessage.unpack(data);

            System.out.println("ORIGINAL DATA: " +
Arrays.toString(incomingMessage.getData()));

            byte[] decryptedData =
config.getClientEncryptor().decryptData(incomingMessage.getData());

            System.out.println("DECRYPTED DATA: " +
Arrays.toString(decryptedData));
        });

        client.connect();
    }
}
```