

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
вычислительной техники

Кафедра
МОиПЭВМ

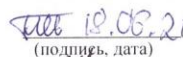
Направление подготовки 09.03.04 «Программная инженерия»

БАКАЛАВРСКАЯ РАБОТА

на тему

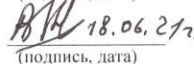
**Сетевой код игрового программного
приложения**

Студент

 18.06.21
(подпись, дата)

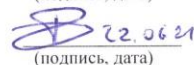
Мельников Глеб Максимович
(ФИО полностью)

Руководитель

 18.06.21
(подпись, дата)

Князев В.Н.
(фамилия, инициалы)

Нормоконтролёр

 22.06.21
(подпись, дата)

Такташкин Д.В.
(фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от _____ № _____)

Заведующий кафедрой

(подпись)

Макарычев П.П.
(фамилия, инициалы)

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____ № _____)

Секретарь ГЭК

(подпись)

Попова Н.А.
(фамилия, инициалы)

Пенза, 2021

Федеральное государственное бюджетное образовательное учреждение
высшего образования

ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет

Вычислительной техники

Кафедра

МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой
П.П. Макарычев
«21» ЯНВАРЯ 2021 г

**ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА**

1. Студент Мельников Глеб Максимович гр. 17ВП1 факультета ВТ
<Фамилия, имя, отчество полностью>

2. Тема работы Сетевой код игрового программного приложения

Тема утверждена приказом ПГУ № 00270/0 от "21" ЯНВАРЯ 2021

3. Руководитель работы к.т.н., доцент кафедры «МОиПЭВМ» Князев В.Н.

4. Задание на работу (назначение разработки, исходные данные и т.п.)

1) *Разработать сетевой код игрового программного приложения для экшен игры.*

Реализовать систему матчмейкинга для игровых сессий со случайными напарниками.

Реализовать уполномоченный, прослушивающий игровой сервер.

2) *Среда выполнения: операционная система Windows 10,*

Языковые средства разработки: C#, UML.

Инструментальные средства: Microsoft Visual Studio, Unity, ReSharper, IIS, PostgreSQL

Draw.io, Microsoft Project

3) *Технологии разработки RUP, ООП, ECS, MVC.*

Основные функции игрового приложения:

а) регистрация новых пользователей;

б) вход пользователей в систему;

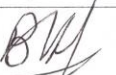
в) однопользовательский режим игры;

г) многопользовательский режим игры со случайным напарником.

5. Перечень подлежащих разработке вопросов:

- 1) Анализ предметной области и постановка задачи;
- 2) Проектирование;
- 3) Реализация;
- 4) Тестирование;
- 5) Оформление пояснительной записки.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области и постановка задачи	15%	14.04.2021– 22.04.2021	
2	Проектирование	25%	23.04.2021– 07.05.2021	
3	Реализация	30%	08.05.2021– 26.05.2021	
4	Тестирование	20%	27.05.2021– 07.06.2021	
5	Оформление пояснительной записки	10%	08.06.2021– 14.06.2021	

Дата выдачи задания "28" декабря 2020 г.

Руководитель бакалаврской работы

 28.12.2020.
(подпись, дата)

Задание к исполнению принял студент

 28.12.20
(подпись, дата)

Работу к защите допустить
Декан факультета

(подпись, дата)

Реферат

Пояснительная записка содержит 131 страницу, 73 рисунка, 14 таблиц, 18 листингов, 32 использованных источников, 8 приложений.

СЕТЕВОЙ КОД, ИГРОВОЕ ПРИЛОЖЕНИЕ, СЕТЕВЫЕ ИГРОВЫЕ ПРИЛОЖЕНИЯ, UDP HOLE PUNCHING, C#, ASP.NET CORE, WEB API, UNITY, ECS, UML, ООП.

Объектом разработки является игровое приложение с возможностью многопользовательского режима игры.

Цель работы – разработка сетевого кода игрового программного приложения, позволяющего игрокам участвовать в игровых сессиях со случайными напарниками.

Результатом разработки является игровое приложение, позволяющее пользователю играть как в одиночном, так и в многопользовательском режиме игры. Для организации многопользовательских игровых сессий был реализован сервер-матчмейкер, ответственный за поиск напарников и создание игровых сессий.

Разработка проводилась на языке программирования C#. В качестве среды разработки использовалась IDE Visual Studio 2019. В качестве СУБД была выбрана PostgreSQL. Игровое приложение было реализовано при помощи средств игрового фреймворка Unity. Реализованное приложение было отлажено и протестировано.

					ПГУ 09.03.04 – 08BP171.14 ПЗ			
Изм.	Лист	№ докум.	Подпись	Дата	Сетевой код игрового программного приложения.	Лит.	Лист	Листов
Разраб.		Мельников Г.М.		18.06.21				
Провер.		Князев В.Н.		18.06.21	Пояснительная записка		3	131
Н. Контр.		Такташкин Д.В.		27.06.21		Группа 17ВП1		
Утверд.		Макарычев П.П.						

Содержание

Введение.....	7
1 Анализ предметной области	10
1.1 Анализ предметной области и постановка задачи.....	10
1.2 Анализ аналогичных решений.....	12
1.2.1 Анализ игры «Salt and Sanctuary».....	13
1.2.2 Анализ игры «Risk of Rain».....	14
1.2.3 Анализ игры «Furi»	17
1.2.4 Результаты анализа программных аналогов	18
1.3 Анализ целевой аудитории.....	19
1.4 Анализ требований.....	22
1.4.1 Функциональные требования.....	22
1.4.2 Нефункциональные требования	26
1.5 Планирование разработки и оценка бюджета.....	27
2 Проектирование сетевого кода	32
2.1 Выбор сетевой архитектуры	32
2.2 Выбор сетевого протокола	36
2.3 Проектирование матчмейкера.....	37
2.3.1 Логическое проектирование базы данных.....	38
2.3.2 Проектирование API	38
2.4 Проектирование клиентской части приложения	40
2.5 Проектирование алгоритмов.....	43
2.5.1 Проектирование алгоритма для сервиса создания матчей	43
2.5.2 Проектирование алгоритма для игрового матча.....	44

	5
2.5.3 Проектирование алгоритма UDP Hole Punching.....	45
2.5.4 Проектирование алгоритма обмена данными	51
2.6 Проектирование формата для обмена данными	52
3 Реализация сетевого кода	56
3.1 Реализация матчмейкера	56
3.1.1 Реализация базы данных.....	56
3.1.2 Реализация API	57
3.2 Реализация клиентской части приложения	58
4 Контроль качества программного обеспечения.....	62
4.1 Функциональное тестирование.....	62
4.1.1 Функциональное тестирование веб-API	62
4.1.2 Функциональное тестирование клиентского приложения	63
4.2 Модульное тестирование.....	64
4.3 Интеграционное тестирование.....	67
4.4 Исследовательское тестирование	68
4.5 Метрики кода.....	69
4.5.1 Цикломатическая метрика Мак-Кейба	71
4.5.2 Метрика Холстеда.....	72
Заключение	74
Список использованных источников	76
Приложение А. Глоссарий	80
Приложение Б. Исходный код веб-API.....	85
Приложение В. Исходный код классов-коннекторов.....	92
Приложение Г. Исходный код библиотеки Network.....	103

Приложение Д. Листинг клиентских классов, ответственных за сетевое взаимодействие.....	109
Приложение Е. Результаты функционального тестирования веб-API.....	113
Приложение Ж. Результаты функционального тестирования клиентского приложения.....	119
Приложение И. Результаты исследовательского тестирования игры	128

Введение

Видеоигры существуют не одно десятилетие. За это время они прошли большой путь от простых аркад на первых игровых автоматах и приставках до высокобюджетных видеоигр, создаваемых сегодня, затраты на производство которых составляют миллионы долларов. Разработка видеоигр давно превратилась в полноценную индустрию, занимающую существенную долю от всей индустрии развлечений. Сегодня рынок видеоигр является одним из самых динамично развивающихся рынков в IT, что делает разработку игр востребованным и актуальным направлением в данной сфере [1].

За всю историю игровой индустрии было создано огромное множество самых разнообразных видеоигр, начиная от текстовых приключенческих игр и заканчивая крупномасштабными action/RPG в открытых мирах. Все это видеоигровое разнообразие можно разделить на три большие группы: однопользовательские игры, многопользовательские игры и игры, поддерживающие как однопользовательский, так и многопользовательский режимы игры. Представители всех этих групп появились на заре игровой индустрии, и, если вопрос о том, что представляет собой однопользовательский режим игры, по сути не стоит (один игрок взаимодействует с игрой посредством устройства ввода-вывода), ситуация с многопользовательским режимом игры не столь однозначна.

Изначально многопользовательский режим был реализован на одном устройстве. Такой подход предполагает, что два или более игроков взаимодействуют с игрой, используя устройства ввода-вывода, при этом обработка игровой логики производится на одном вычислительном устройстве. В качестве примера ранних игр с реализацией многопользовательского режима на одном устройстве можно назвать «Tennis for Two» и «Pong». Такой подход к организации многопользовательского режима не редко реализуют и в современных видеоиграх, так как он может удачно ложиться на игровой процесс (например, в спортивных симуляторах или файтингах).

Однако уже на заре игровой индустрии был реализован и другой вариант многопользовательского режима, поддерживающий взаимодействие игроков через вычислительную сеть (сетевые игры). При такой организации взаимодействия игроков, каждый участник использует отдельное вычислительное устройство для взаимодействия с игрой. Каждое устройство подключено к вычислительной сети и использует ее для передачи информации, что позволяет актуализировать и синхронизировать состояния игрового мира на всех устройствах. «Maze Ware», «Empire», «Spasim», являются примерами ранних многопользовательских игр, использующих такой принцип. Однако в те годы доступ к терминалам и компьютерам, подключенным к вычислительным сетям, имел крайне ограниченный круг людей. Основной платформой для таких многопользовательских игр были вычислительные сети университетов и исследовательских центров (например, система электронного обучения PLATO), пользователями которых были сотрудники университетов, студенты, школьники [2 – 3]. Позднее, с увеличением доступности персональных компьютеров, развитием и распространением интернета, возможность для игры по сети появлялась у все большего числа игроков, что стало серьезным толчком для развития сетевых игр [3].

Сегодня многопользовательские игры стали трендом. Практически все проекты, находящиеся в двадцатке самых популярных игр на персональных компьютерах по состоянию на апрель 2021 года, являются многопользовательскими [4]. Успешный многопользовательский проект с активной поддержкой может длительное время сохранять интерес игроков и стабильно приносить доход, что является серьезным стимулом для крупных игровых компаний создавать свои онлайн-проекты. Таким образом, разработка многопользовательских игр является перспективной и востребованной задачей.

Создание многопользовательских игр имеет свои особенности и ограничения. Разработчикам приходится решать целый комплекс технических

проблем, чтобы предоставить игрокам возможность совместной игры. Программные средства, реализованные в рамках игрового приложения, и обеспечивающие сетевое взаимодействие называют сетевым кодом. Сетевой код является неотъемлемой частью любой сетевой игры. От того, насколько удачно он реализован напрямую зависит получаемый игровой опыт, что делает разработку сетевого кода важнейшей задачей при разработке многопользовательской игры.

Исходя из вышесказанного, можно сделать вывод, что цель выпускной квалификационной работы, заключающаяся в разработке сетевого кода игрового программного приложения, является востребованной и актуальной.

1 Анализ предметной области

1.1 Анализ предметной области и постановка задачи

Игровая индустрия является одной из самых быстро развивающихся отраслей в IT. По данным Newzoo в 2020 году 2.7 миллиарда игроков по всему миру потратило 159.3 млрд долларов на игры, а к 2023 году рынок видеоигр превысит 200 млрд долларов [5]. Уже сейчас рынок видеоигр оставил позади ряд других конкурентных рынков, в частности кино- и музыкальную индустрию. На рисунке 1 представлены доходы игровой, кино- и музыкальной индустрии за 2019 год [6].

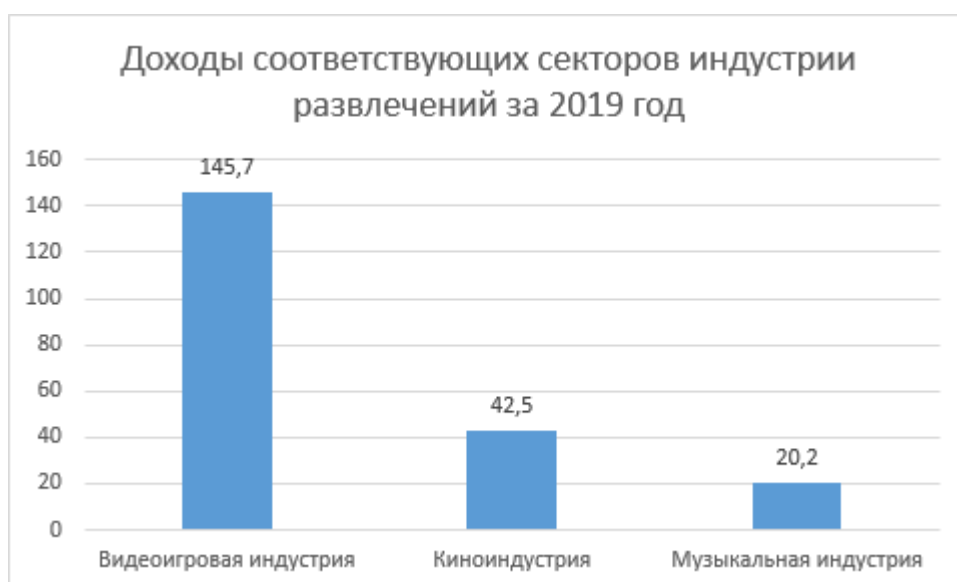


Рисунок 1 – Доходы игровой, кино- и музыкальной индустрии за 2019 год

По данным, представленным на диаграмме видно, что доходы игровой индустрии в 2019 году превысили более чем в 2 раза доход от кино- и музыкальной индустрии вместе взятых. Данная статистика позволяет сделать вывод, что индустрия видеоигр активно развивается и растет, и в перспективе данный тренд сохранится. Это делает разработку видеоигр сегодня крайне перспективной.

Заметную часть дохода в игровой индустрии приносят многопользовательские проекты. Они пользуются популярностью у обширной аудитории игроков. Это связано, с одной стороны, с возможностью игроков

взаимодействовать друг с другом в рамках игрового процесса, что позволяет создавать принципиально иной игровой опыт, не доступный однопользовательским играм, с другой стороны, такие игры часто имеют привлекательную для игроков модель распространения. Не редко многопользовательские игры распространяются по модели Free-to-play, что делает их значительно более доступными, чем однопользовательские проекты, которые обычно распространяются по фиксированной цене. Кроме того, успешные многопользовательские проекты как правило имеют длительную историю поддержки со стороны разработчиков, которые стараются актуализировать игру и добавлять в нее новые игровые возможности. Со стороны разработчиков и издателей многопользовательские проекты интересны как источники стабильной прибыли, получаемой, чаще всего, за счет продажи игровых дополнений, внутриигровых предметов или подписочной системы распространения. В качестве примеров успешных многопользовательских проектов можно назвать такие игры как «Fortnite», «GTA V», «Counter-Strike: Global Offensive» и др. [7 – 9].

Многопользовательские игры могут предложить разные способы взаимодействия игроков. Часто они имеют соревновательный характер. Однако помимо соревновательных игр существует и другой вид многопользовательских игр, предполагающий не соперничество между игроками, а их совместную работу для достижения победы. Такие игры называют кооперативными. Часто кооперативные игры характеризуются сложным геймплеем, что требует от игроков слаженной и обдуманной командной работы. Успешные кооперативные игры с активной поддержкой со стороны разработчиков или комьюнити остаются востребованными длительное время. Примерами успешных кооперативных игр являются «Portal 2», серия игр «Bordarlands», «It Takes Two» и др. [10 – 12].

В рамках выпускной квалификационной работы необходимо разработать сетевой код для кооперативного игрового приложения. В общем случае под понятием «сетевой код» понимают все аспекты игрового

приложения, так или иначе связанные с многопользовательским режимом игры. Объединение нескольких игроков в рамках одной игровой сессии, организация обмена данными между этими игроками, оптимизация сетевого трафика, компенсация задержек, механизмы предсказаний, репликация объектов, выбор сетевой архитектуры – все это проблемы, решаемые при разработке сетевого кода.

При реализации сетевого кода важно учитывать особенности игры, для которой он разрабатывается. При этом следует иметь ввиду, что игровой код может зависеть от сетевого, однако обратная ситуация недопустима, так как может сделать процесс развития игры практически невозможным [2]. Именно поэтому разработка игры велась в тесном сотрудничестве с членом команды, ответственным за геймдизайн.

Разрабатываемая игра представляет собой экшен. Геймплей игры построен вокруг сражений с боссом (особенно сложным персонажем-противником). Игра должна предоставлять возможность играть как в однопользовательском режиме игры, так и в многопользовательском. Многопользовательский режим позволяет двум случайным игрокам совместно сразиться с боссом в рамках одной игровой сессии. Сетевой код должен учитывать динамичную природу экшен-игр и реализовывать эффективный способ обмена данными для своевременной актуализации состояния игрового мира. Игра должна поддерживать систему учетных записей, позволяющую однозначно идентифицировать каждого игрока.

1.2 Анализ аналогичных решений

Сегодня существует множество игр, являющихся кооперативными, или имеющие возможности для кооперативной игры, предлагающих игрокам совместно сражаться с боссами. Произведем сравнительный анализ некоторых из них.

1.2.1 Анализ игры «Salt and Sanctuary»

Игра Salt and Sanctuary разработана студией Ska Studios и выполнена в жанре action/RPG с элементами платформера. Salt and Sanctuary испытала на себе сильное влияние игр серии Dark Souls, получивших свою известность за сложный, но одновременно честный по отношению к игроку геймплей. Она активно использует и развивает идеи, заложенные в данной серии игр. На рисунке 2 представлен скриншот из игры, демонстрирующий сражение с боссом.



Рисунок 2 – Скриншот из игры Salt and Sanctuary

С точки зрения игрового процесса Salt and Sanctuary предлагает игроку исследовать игровой мир, представляющий собой набор связанных друг с другом локаций, заполненных противниками. Помимо противников на локациях могут находиться святилища, выполняющие роль контрольных точек, предметы, нейтральные по отношению к игроку неигровые персонажи. Как правило, в конце каждой локации находится босс. После победы над боссом локация считается пройденной, и игрок может преступить к изучению следующей локации.

Основная часть геймплея игры Salt and Sanctuary построена вокруг сражений. Благодаря комплексной, но при этом удобной и понятной боевой системе, а также грамотному подходу разработчиков к созданию боевых ситуаций и дизайну противников, каждое боевое столкновение в игре представляет собой тактическую задачу, требующую от игрока внимательности, грамотного позиционирования и решительности.

Отдельного внимания заслуживают сражения с боссами. Как правило, каждый босс является уникальным противником, со своим особенным поведением и набором боевых приемов. Сражение с боссом требует от игрока значительно больше усилий, однако и награда за победу соответствующая. Кроме того, именно победы над боссами дают возможность продвинуться дальше в прохождении игры. Сражения с боссами являются главными игровыми испытаниями, предлагающими наиболее сложный, напряженный и интересный геймплей.

Игра Salt and Sanctuary имеет многопользовательский режим, однако реализован он на одном устройстве. Игра по сети не предусмотрена [13].

Таким образом, можно сделать вывод, что игра Salt and Sanctuary отличается комплексным геймплеем и проработанной боевой системой, однако режим многопользовательской игры по сети в ней отсутствует. Кроме того, перед сражениями с боссами локаций игрок вынужден проходить, непосредственно, сами локации, что может показаться скучным и рутинным занятием для игроков, пренебрегающих исследовательской частью игрового процесса.

1.2.2 Анализ игры «Risk of Rain»

Игра Risk of Rain разработана компанией Noroo Games и выполнена в жанре экшен платформера с элементами roguelike. На рисунке 3 представлен скриншот из игры, демонстрирующий сражение с боссом.



Рисунок 3 – Скриншот из игры Risk of Rain

Перед началом игры игрок должен выбрать одного из доступных персонажей. По умолчанию доступен один персонаж, однако в процессе прохождения открываются и другие. Каждый персонаж обладает уникальным набором характеристик и особенностей.

Цель игрока на каждом уровне (кроме последнего) является поиск телепорта, который появляется в случайном месте карты. В процессе поиска игрок вынужден сражаться с постоянно появляющимися противниками, которые после своей смерти оставляют внутриигровые деньги и опыт. Опыт нужен для повышения уровня персонажа. Каждое повышение уровня увеличивает количество очков здоровья и урон героя. Деньги же нужны для покупки предметов, открытия сундуков, молитв у алтарей и т.д. В игре имеется более 110 предметов, открываемых в процессе игры. Они позволяют усилить атакующие или защитные способности персонажа. Предметы делятся на пассивные и активные. Пассивные предметы дают постоянный бонус, в то время как активные добавляют способность персонажу и имеют время перезарядки. Персонаж может иметь сколько угодно пассивных предметов (в

том числе и повторяющихся, так как их бонус складывается), однако активный предмет позволено иметь всего один.

Найдя телепорт, игрок может его активировать, после чего в течение определенного времени (90 или 120 секунд в зависимости от сложности) игроку необходимо выживать. При этом врагов становится значительно больше и на карте появляется босс. После того, как время на таймере закончится, новые противники перестанут появляться. Игрок должен уничтожить оставшихся противников, после чего у него появляется возможность перейти на следующий уровень. При переходе на следующий уровень деньги игрока переводятся в опыт.

На предпоследнем уровне у игрока появляется выбор. Он может отправиться на последний уровень, чтобы сразиться с финальным боссом, либо он может перейти на усложненную версию первого уровня, сохранив при этом все свои предметы. Выбрав первую опцию, игрок получает возможность пройти игру, победив финального босса. Выбрав вторую опцию, игрок может дополнительно усилить своего персонажа перед финальным сражением, проходя игру заново. Добравшись до предпоследнего уровня, игрок может перейти на финальный уровень после любого последующего.

Кроме того, в игре реализована механика повышения сложности в зависимости от времени таймера. Чем больше игрок провел времени в очередной игре, тем она становится сложнее [14].

Игра Risk of Rain является типичным представителем жанра roguelike. Она обладает целым комплексом сложных и взаимосвязанных игровых механик. Полное прохождение потребует от игрока не один десяток попыток, в ходе которых он должен будет изучить тонкости и нюансы игрового процесса.

Несмотря на то, что в игре есть сражения с боссами, геймплей игры не сосредоточен на них. Игрок в большей степени озадачен не победой над очередным противником, а обдумыванием оптимальной стратегии реализации имеющихся у него ресурсов. Несмотря на то, что игра предлагает

комплексный геймплей, она может не подойти широкому кругу игроков из-за высокого порога вхождения.

Игра поддерживает как локальный многопользовательский режим, так и многопользовательский режим по сети, однако последний может потребовать от пользователей технических навыков. Из-за того, что у игры нет центрального сервера, который бы мог обеспечить соединение игроков, одному из пользователей может потребоваться пробросить порт на своем маршрутизаторе, после чего ему необходимо узнать свой глобальный ip и сообщить эту информацию своим сокомандникам.

1.2.3 Анализ игры «Furi»

Игра Furi разработана независимой студией The Game Bakers. С точки зрения жанра игру можно отнести к экшен играм с элементами shoot'em up, hack and slash и bullet hell. На рисунке 4 представлен скриншот, демонстрирующий битву с одним из боссов игры.



Рисунок 4 – Скриншот из игры Furi

Геймплейно Furi предлагает игрокам последовательно сразиться с несколькими боссами. Каждый босс имеет свои уникальные приемы,

особенности и множество фаз. Фазы могут значительно отличаться друг от друга, заставляя игрока постоянно менять тактику и методы сражения. После победы над боссом прогресс игрока сохраняется, и игрок может приступить к сражению со следующим боссом не опасаясь оказаться в самом начале игры после поражения. Игра полностью сосредотачивается на боевой системе и сражениях с боссами и не предлагает игроку других игровых активностей [15].

Геймплейно игра полностью построена вокруг сражений с боссами, однако она имеет существенный недостаток. Единственный доступный режим игры – одиночный, следовательно, играть в кооперативе с другими игроками не представляется возможным.

1.2.4 Результаты анализа программных аналогов

На основании проведенного анализа можно сделать вывод, что несмотря на наличие аналогичных приложений, разработка собственного решения остается актуальной.

Игра *Salt and Sanctuary* предлагает комплексный и продуманный геймплей, однако в ней не реализована возможность многопользовательской игры по сети (только локальный многопользовательский режим). Кроме того, хоть игра и делает серьезный акцент на сражениях с боссами, они не составляют основу всего игрового процесса.

Игра *Risk of Rain* предлагает сложный игровой процесс с традиционным для игр жанра *roguelike* несколько репитативным игровым процессом, требующим от игрока совершать не один десяток попыток для прохождения. Несмотря на то, что в игре реализованы сражения с боссами, они не являются ключевым элементом геймплея. Игра имеет локальный и сетевой многопользовательский режим, однако для игры в последнем от игроков могут потребоваться технические навыки.

Игра *Furi* практически полностью сосредотачивается на сражениях с боссами, однако она не реализует многопользовательский режим, не позволяя нескольким игрокам проходить игру совместно.

Результаты сравнения аналогов представлены в таблице 1.

Таблица 1 – Результаты сравнения аналогов

Характеристика	Salt and Sanctuary	Risk of Rain	Furi	Свое решение
Сражения с боссами составляют основу геймплея.	Сражения с боссами составляют заметную часть геймплея, однако игровой процесс подразумевает множество других активностей.	Сражения с боссами не являются ключевым элементом геймплея.	Да.	Да.
Наличие сетевого многопользовательского режима.	Нет.	Да, однако он может потребовать от игроков технических навыков.	Нет.	Да.

Таким образом было принято решение разработать собственную кооперативную игру с сетевым многопользовательским режимом, основу геймплея которой будут составлять битвы с боссами.

1.3 Анализ целевой аудитории

Основными игровыми платформами на сегодняшний день являются персональные компьютеры, консоли и смартфоны. По данным Statista на март 2019 года данные платформы являются наиболее перспективными, с точки зрения дальнейшего роста (рисунок 5) [16].

Следовательно, разрабатывая игровой проект, целесообразно ориентироваться именно на эти платформы. Стоит учитывать, что смартфоны как игровая платформа вносят серьезные технические ограничения, связанные с относительно невысокими вычислительными возможностями данных устройств. Кроме того, возникают дополнительные трудности при реализации сетевых игр, так как игроки часто используют смартфоны как игровую платформу в дороге, что может негативно сказываться на качестве интернет соединения. Консоли и персональные компьютеры, как правило, не имеют такой проблемы.

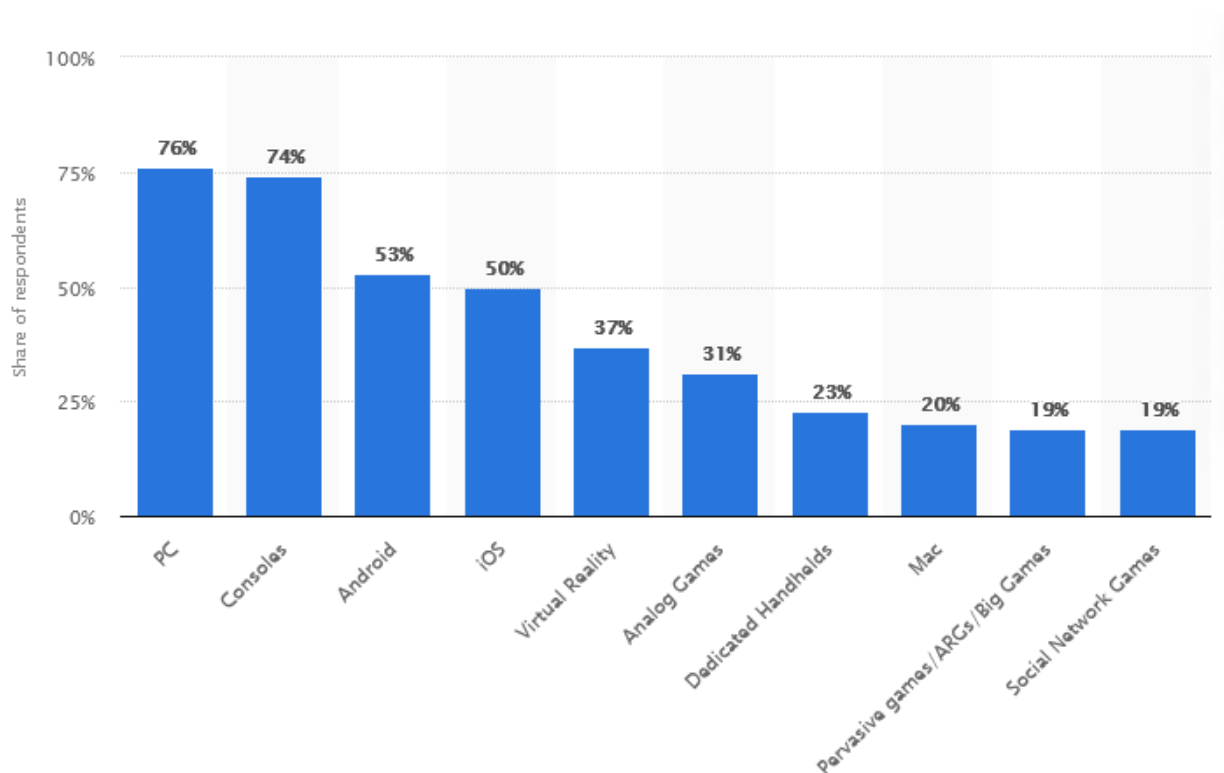


Рисунок 5 – Наиболее перспективные игровые платформы с точки зрения их дальнейшего роста для разработчиков игр

На основании представленных данных было принято решение выбрать в качестве целевой платформы персональные компьютеры, так как данная платформа является одной из самых популярных и не накладывает жестких технических ограничений на разработку.

Большая часть игроков, использующих персональный компьютер в качестве игровой платформы, выбирают для игр ОС Windows. На рисунке 6 представлена диаграмма, демонстрирующая распределение пользователей по разным ОС согласно данным сервиса цифрового распространения компьютерных игр Steam на октябрь 2020 года [17].

По данной диаграмме можно сделать вывод, что ОС Windows 10 значительно опережает другие ОС по популярности, в качестве ОС для игр. Таким образом, было принято решение разрабатывать игровое приложение для персональных компьютеров, работающих под управлением ОС Windows 10.

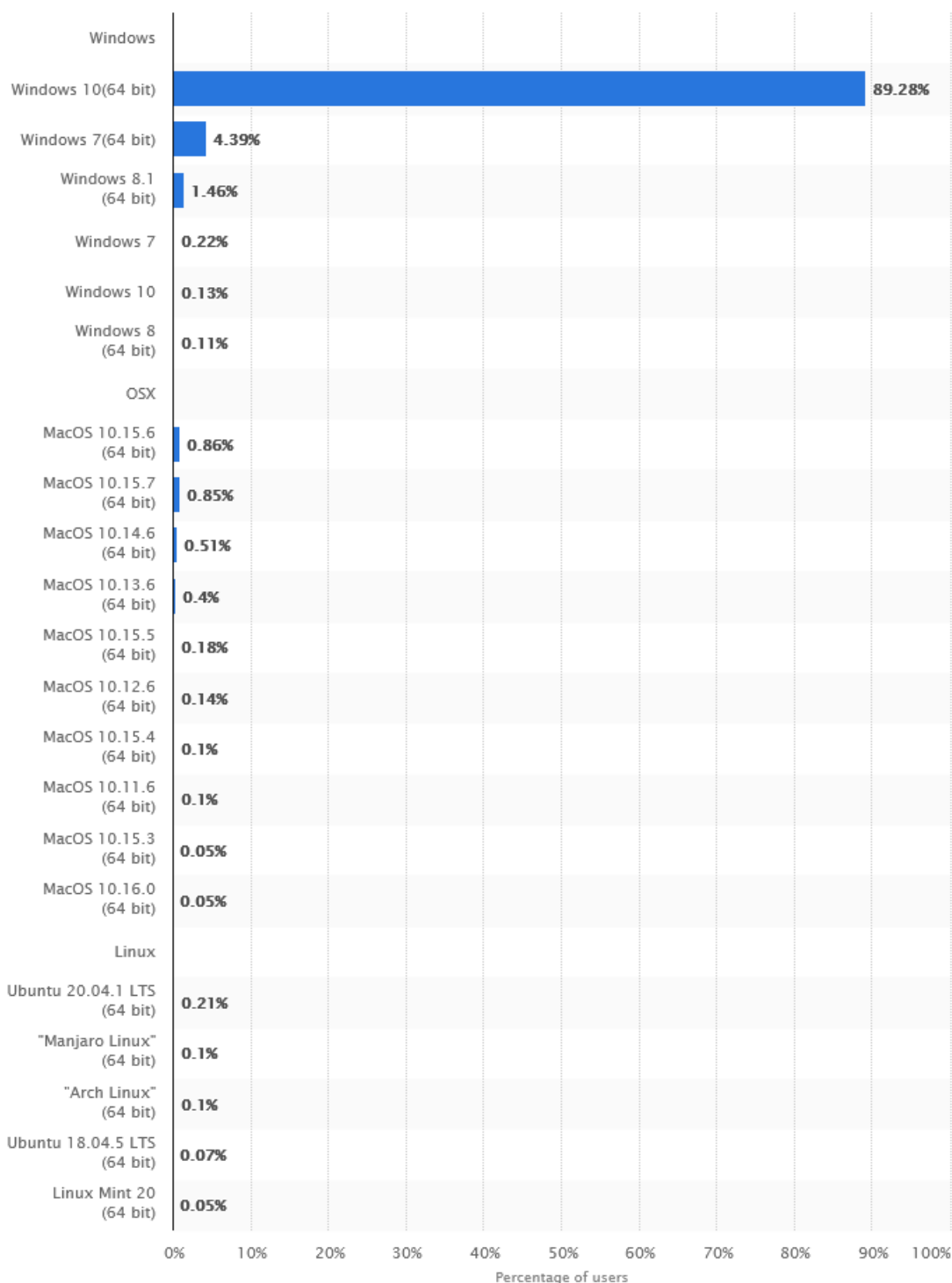


Рисунок 6 – Распределение пользователей сервиса Steam по разным ОС

Однако популярность смартфонов и игровых консолей в качестве платформ для видеоигр становится серьезным аргументом для того, чтобы заложить при разработке возможности для дальнейшего переноса разрабатываемой игры на эти платформы.

1.4 Анализ требований

1.4.1 Функциональные требования

В результате анализа предметной области были построены диаграммы прецедентов, описывающие функциональные требования, предъявляемые к разрабатываемой системе.

На рисунке 7 представлена укрупненная диаграмма прецедентов, описывающая всю систему в целом.



Рисунок 7 – Диаграмма прецедентов игрового приложения

На диаграмме представлено два актера. Приведем их описание.

Актер «Игрок» представляет собой игрока, использующего игровое приложение.

Актер «Сервер» представляет собой серверную часть игрового приложения, ответственную за организацию игровых матчей.

На рисунке 8 представлена диаграмма прецедентов игровой подсистемы разрабатываемого приложения. Под игровой подсистемой в данном случае понимается та часть приложения, которая ответственна за обеспечение игрового процесса.

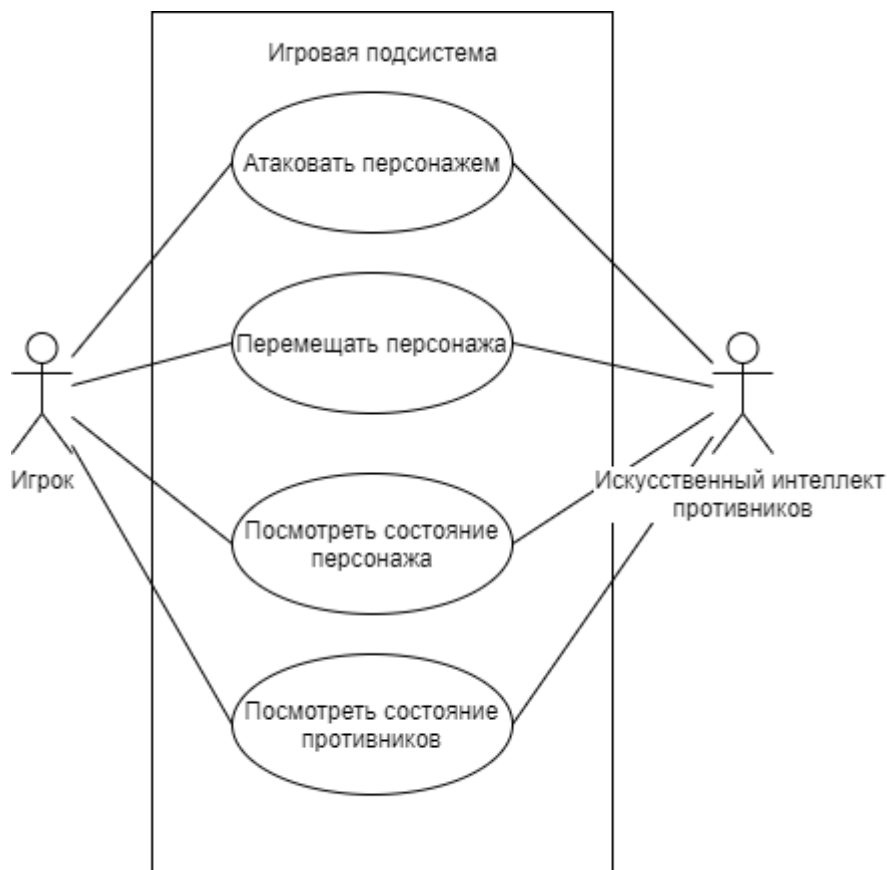


Рисунок 8 – Диаграмма прецедентов игровой подсистемы

В контексте данной диаграммы представлен актер «Искусственный интеллект противников», ответственный за поведение игровых противников и реагирующий на действия игрока.

Диаграмма на рисунке 8 актуальна как для однопользовательского, так и для многопользовательского режима игры. Единственное существенное отличие этих двух режимов – количество участвующих в игровой сессии игроков.

На рисунке 9 представлена диаграмма прецедентов серверной части приложения.

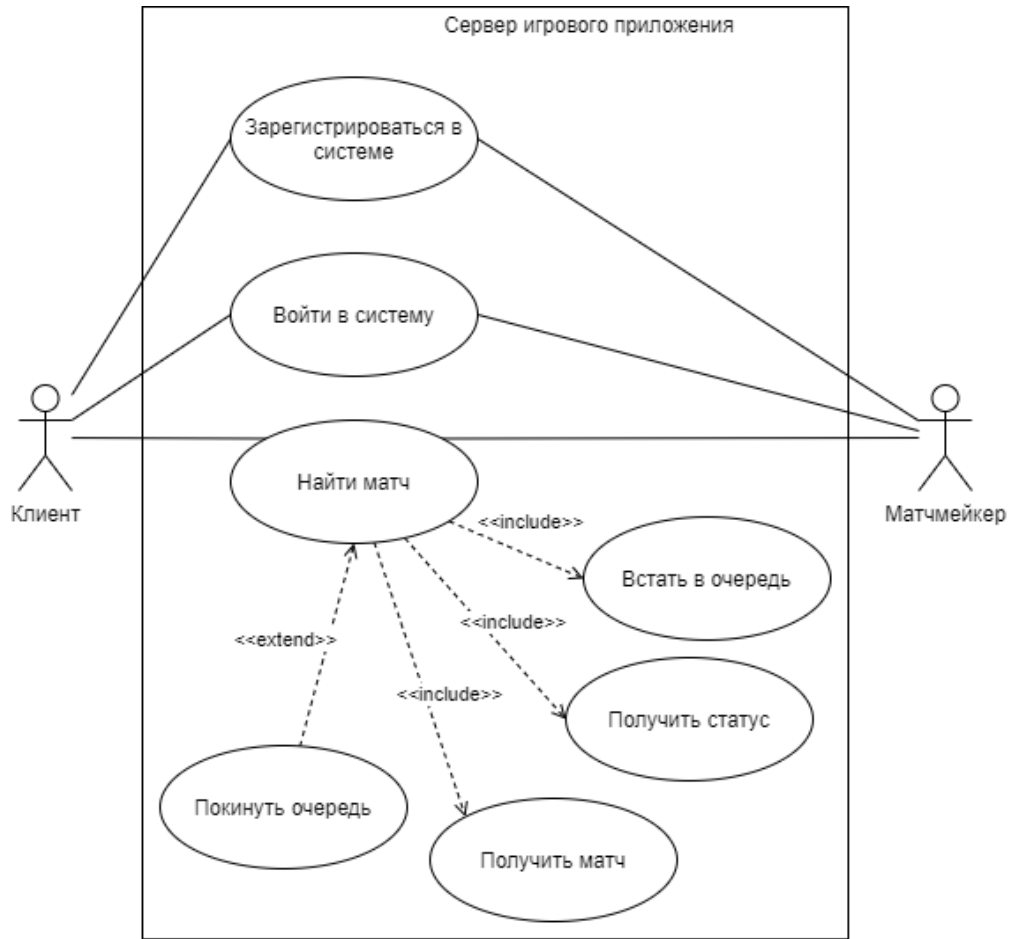


Рисунок 9 – Диаграмма прецедентов серверной части приложения

Актёр «Матчмейкер» представляет собой компонент приложения, ответственный за организацию игровых сессий.

Актёр «Клиент» представляет собой в данном случае пользователя серверного API.

Рассмотрим спецификацию некоторых прецедентов серверной части игрового приложения. На рисунке 10 представлена спецификация прецедента «Найти матч».

Прецедент: Найти матч
ID: 1
Краткое описание: Клиент ищет команду для совместной игры.
Главные актёры: Клиент, Матчмейкер.
Предусловия: Клиент вошёл в систему.
Основной поток: 1. Include(Встать в очереди). 2. Пока клиент не добавлен в матч 2.1. Include(Получить статус). 3. Include(Получить матч).
Постусловия: Клиент получил информацию о матче, в котором участвует.

Рисунок 10 – Спецификация прецедента «Найти матч»

На рисунке 11 представлена спецификации прецедента «Встать в очередь».

Прецедент: Встать в очередь
ID: 2
Краткое описание: Клиент встает в очередь на матч.
Главные актеры: Клиент, Матчмейкер.
Предусловия: Клиент вошел в систему.
Основной поток: <ol style="list-style-type: none"> 1. Клиент совершает запрос на постановку в очередь. 2. Сервер определяет конечные точки клиента. 3. Если клиент уже находится в очереди <ol style="list-style-type: none"> 3.1. Сервер сообщает клиенту о том, что он уже находится в очереди. 4. Иначе <ol style="list-style-type: none"> 4.1. Сервер сообщает клиенту о том, что клиент добавлен в очередь.
Постусловия: Клиенту отправлен ответ.

Рисунок 11 – Спецификация прецедента «Встать в очередь»

На рисунке 12 представлена спецификация прецедента «Получить статус».

Прецедент: Получить статус
ID: 3
Краткое описание: Клиент проверяет свой статус.
Главные актеры: Клиент, Матчмейкер.
Предусловия: Клиент вошел в систему.
Основной поток: <ol style="list-style-type: none"> 1. Клиент совершает запрос своего статуса. 2. Если клиент находится в очереди <ol style="list-style-type: none"> 2.1. Сервер сообщает о том, что клиент находится в состоянии ожидания. 3. Иначе, если клиент был добавлен в матч <ol style="list-style-type: none"> 3.1. Сервер сообщает о том, что клиент был добавлен в матч. 4. Иначе <ol style="list-style-type: none"> 4.1. Сервер сообщает о том, что клиент отсутствует.
Постусловия: Клиент узнает свой статус.

Рисунок 12 – Спецификация прецедента «Получить статус»

На рисунке 13 представлена спецификация прецедента «Получить матч».

Прецедент: Получить матч
ID: 4
Краткое описание: Клиент получает информацию о матче, в котором участвует.
Главные актеры: Клиент, Матчмейкер.
Предусловия: Клиент вошел в систему.
Основной поток: <ol style="list-style-type: none"> 1. Клиент совершает запрос на получение матча. 2. Если клиент был добавлен матч <ol style="list-style-type: none"> 2.1. Сервер сообщает информацию о матче. 3. Иначе <ol style="list-style-type: none"> 3.1. Сервер сообщает о том, что клиент не участвует ни в одном матче.
Постусловия: Клиент получил информацию о матче, в котором участвует.

Рисунок 13 – Спецификация прецедента «Получить матч»

На рисунке 14 представлена спецификация прецедента «Покинуть очередь».

Прецедент: «Покинуть очередь»
ID: 5
Краткое описание: Клиент покидает очередь на матч.
Главные актеры: Клиент, Матчмейкер.
Предусловия: Клиент вошел в систему.
Основной поток: 1. Клиент совершает запрос на выход из очереди на матч. 2. Если клиент находится в очереди 2.1. Сервер удаляет клиента из очереди.
Постусловия: Клиент удален из очереди на матч.

Рисунок 14 – Спецификация прецедента «Покинуть очередь»

Диаграмма последовательности, поясняющая прецедент «Найти матч» представлена на рисунке 15.

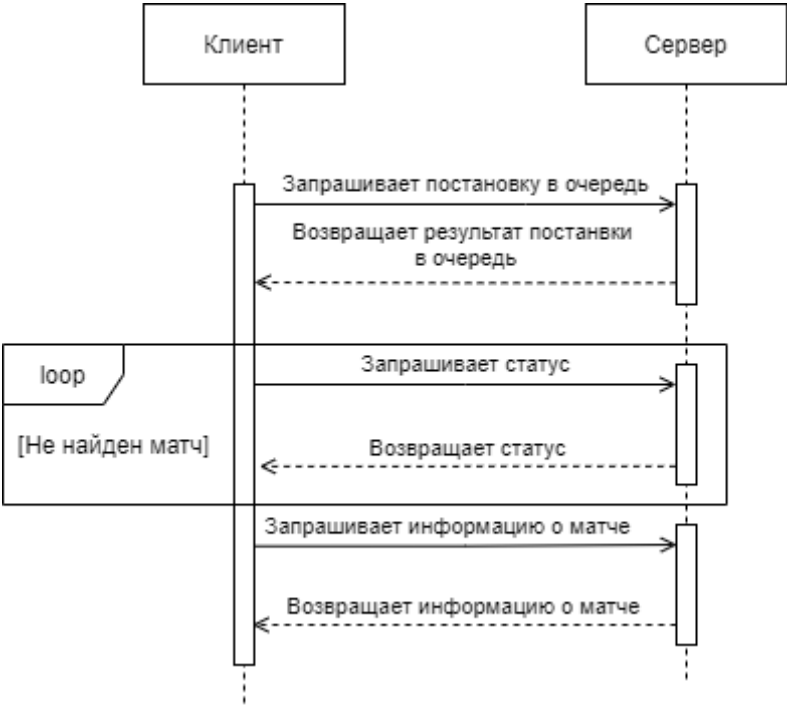


Рисунок 15 – Диаграмма последовательности для прецедента «Найти матч»

Данная диаграмма демонстрирует механизм определения клиентами их статуса в процессе поиска матча. Клиентам необходимо совершать периодические запросы, чтобы узнать о своем участии в матче.

1.4.2 Нефункциональные требования

Серверная часть приложения должна быть выполнена на языке C# с использованием фреймворка ASP.NET Core. Необходимо реализовать

web-сервис, позволяющий игрокам связываться друг с другом в рамках одной игровой сессии. В качестве СУБД необходимо использовать PostgreSQL. Для работы с базой данных необходимо использовать ORM Entity Framework Core. Для сетевого взаимодействия игроков необходимо реализовать собственное решение поверх протокола UDP.

Клиентская часть приложения должна быть реализована на языке C# при помощи средств игрового фреймворка Unity 2019. Игра должна корректно работать на персональных компьютерах под управлением ОС Windows 10.

При разработке необходимо использовать IDE Visual Studio 2019 с установленным плагином ReSharper.

1.5 Планирование разработки и оценка бюджета

Для определения стоимости и трудоемкости проекта произведем структурное планирование. В таблице 2 представлен перечень задач проекта.

Таблица 2 – Перечень задач проекта

№	Название задачи	Длительность	Предшественники
1	Начало проекта	-	-
2	Анализ требований	-	-
3	Анализ аналогичных решений	3 дня	1
4	Анализ сетевых архитектур в многопользовательских играх	3 дня	3
5	Анализ игрового приложения	3 дня	3
6	Выбор оптимальной сетевой архитектуры	1 день	4, 5
7	Анализ требований завершен	-	6
8	Выбор средств реализации	-	-
9	Анализ возможных программных средств	2 дня	1
10	Выбор оптимальных программных средств	1 день	9
11	Выбор средств завершен	-	10
12	Реализация проекта	-	-
13	Разработка структур данных	2 дня	7, 11
14	Разработка API	1 день	7, 11
15	Реализация серверной части	25 дней	13, 14
16	Реализация сетевого кода на клиентской части	25 дней	15
17	Оптимизация сетевого трафика	20 дней	16
18	Реализация завершена	-	17
19	Отладка	-	-
20	Тестирование серверной части	7 дней	18
21	Тестирование клиентского сетевого кода	7 дней	18
22	Совместное тестирование серверной и клиентской части	7 дней	20, 21
23	Исправление ошибок	3 дня	22
24	Отладка завершена	-	23
25	Конец проекта	-	24

В таблице 3 приведено описание ресурсов, необходимых для реализации проекта.

Таблица 3 – Список ресурсов проекта

Название ресурса	Тип	Стандартная ставка	Ставка сверхурочных
Программист 1	Трудовой	300 р/ч	400 р/ч
Программист 2	Трудовой	300 р/ч	400 р/ч
Интернет	Затраты		

Назначение ресурсов на задачи представлено в таблице 4.

Таблица 4 – Назначение ресурсов

№	Название задачи	Ресурсы	Единицы
1	Начало проекта		
2	Анализ требований		
3	Анализ аналогичных решений	Программист 1 Интернет	50%
4	Анализ сетевых архитектур в многопользовательских играх	Программист 1 Интернет	50%
5	Анализ игрового приложения	Программист 1 Программист 2 Интернет	50% 50%
6	Выбор оптимальной сетевой архитектуры	Программист 1 Программист 2 Интернет	50% 50%
7	Анализ требований завершен		
8	Выбор средств реализации		
9	Анализ возможных программных средств	Программист 1 Программист 2 Интернет	50% 50%
10	Выбор оптимальных программных средств	Программист 1 Программист 2 Интернет	50% 50%
11	Выбор средств завершен		
12	Реализация проекта		
13	Разработка структур данных	Программист 1 Интернет	75%
14	Разработка API	Программист 1 Интернет	25%
15	Реализация серверной части	Программист 1 Интернет	100%
16	Реализация сетевого кода на клиентской части	Программист 1 Программист 2 Интернет	75% 25%
17	Оптимизация сетевого трафика	Программист 1 Интернет	100%
18	Реализация завершена		
19	Отладка		
20	Тестирование серверной части	Программист 1 Интернет	50%

Продолжение таблицы 4

№	Название задачи	Ресурсы	Единицы
21	Тестирование клиентского сетевого кода	Программист 1 Программист 2 Интернет	50% 50%
22	Совместное тестирование серверной и клиентской части	Программист 1 Программист 2 Интернет	100% 50%
23	Исправление ошибок	Программист 1 Программист 2 Интернет	100% 25%
24	Отладка завершена		
25	Конец проекта		

Внесем данные проекта в систему управления проектами Microsoft Project (рисунок 16).

		Режим задачи ▼	Название задачи ▼	Длительность ▼	Начало ▼	Окончание ▼	Предшественники ▼	Названия ресурсов ▼
1			Начало проекта	0 дней	Пн 08.02.21	Пн 08.02.21		
2			▲ Анализ требований	7 дней	Пн 08.02.21	Пн 15.02.21		
3			Анализ аналогичных решений	3 дней	Пн 08.02.21	Ср 10.02.21	1	Программист 1[50%
4			Анализ сетевых архитектур в многопользовательских играх	3 дней	Чт 11.02.21	Сб 13.02.21	3	Программист 1[50%]
5			Анализ игрового приложения	3 дней	Чт 11.02.21	Сб 13.02.21	3	Программист 1[50%
6			Выбор оптимальной сетевой архитектуры	1 день	Сб 13.02.21	Пн 15.02.21	4;5	Программист 1[50%]
7			Анализ требований завершен	0 дней	Пн 15.02.21	Пн 15.02.21	6	
8			▲ Выбор средств реализации	3 дней	Пн 08.02.21	Ср 10.02.21		
9			Анализ возможных программных средств	2 дней	Пн 08.02.21	Вт 09.02.21	1	Программист 1[50%
10			Выбор оптимальных программных средств	1 день	Ср 10.02.21	Ср 10.02.21	9	Интернет[480,00 €]
11			Выбор средств завершен	0 дней	Ср 10.02.21	Ср 10.02.21	10	
12			▲ Реализация проекта	72 дней	Пн 15.02.21	Сб 08.05.21		
13			Разработка структур данных	2 дней	Пн 15.02.21	Ср 17.02.21	7;11	Интернет;Программист 1[50%]
14			Разработка API	1 день	Пн 15.02.21	Вт 16.02.21	7;11	Интернет;Программист 1[50%]
15			Реализация серверной части	25 дней	Ср 17.02.21	Чт 18.03.21	13;14	Интернет;Программист 1[50%]
16			Реализация сетевого кода на клиентской части	25 дней	Чт 18.03.21	Чт 15.04.21	15	Программист 2[25%]
17			Оптимизация сетевого трафика	20 дней	Чт 15.04.21	Сб 08.05.21	16	Программист 1;Интернет
18			Реализация завершена	0 дней	Сб 08.05.21	Сб 08.05.21	17	
19			▲ Отладка	17 дней	Сб 08.05.21	Пт 28.05.21		
20			Тестирование серверной части	7 дней	Сб 08.05.21	Пн 17.05.21	18	Программист 1[50%]
21			Тестирование клиентского сетевого кода	7 дней	Сб 08.05.21	Пн 17.05.21	18	Программист 1[50%]
22			Совместное тестирование серверной и клиентской части	7 дней	Пн 17.05.21	Вт 25.05.21	20;21	Интернет; Программист 1;Программист 2[25%]
23			Исправление ошибок	3 дней	Вт 25.05.21	Пт 28.05.21	22	Интернет;Программист 1[50%]
24			Отладка завершена	0 дней	Пт 28.05.21	Пт 28.05.21	23	
25			Конец проекта	0 дней	Пт 28.05.21	Пт 28.05.21	24	

Рисунок 16 – Список задач проекта

На рисунке 17 представлена полученная средствами среды Microsoft Project диаграмма Ганта.

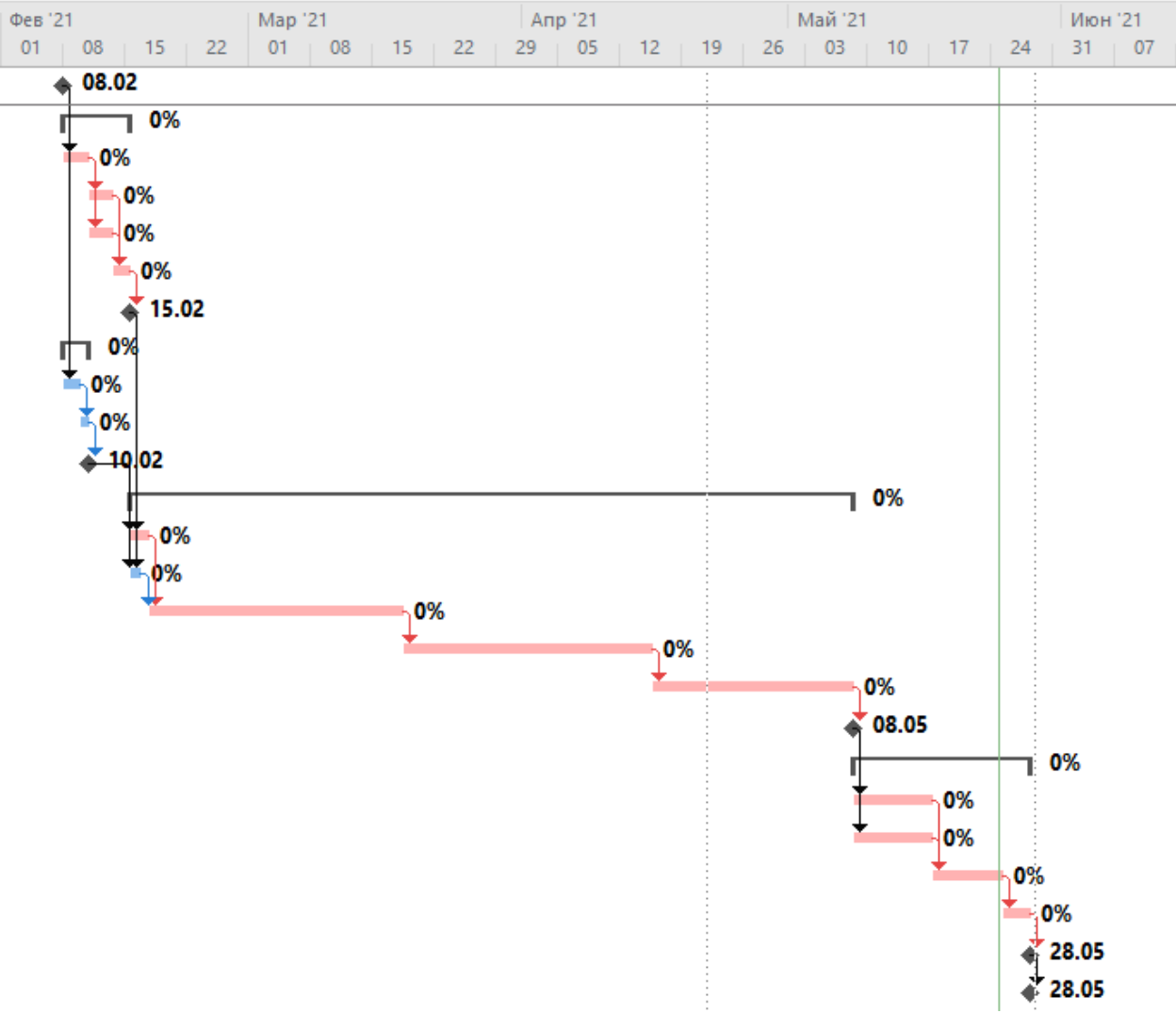


Рисунок 17 – Диаграмма Ганта

По данной диаграмме можно сделать вывод, что критический путь проходит через задачи 1, 3, 4, 5, 6, 7, 13, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25. На рисунке 18 представлен отчет по проекту, полученный средствами Microsoft Project.

Начало	Окончан ие	Длитель ность	Затраты	Трудоза траты	Сверхур очные трудозат раты
Пн 08.02.21	Пт 28.05.21	96 дней	236 880,00 Р	788 ч	0 ч

Рисунок 18 – Отчет по проекту

Согласно полученным данным на реализацию проект необходимо 96 дней. Затраты на проект составляют 236 880 руб. Сверхурочные трудозатраты отсутствуют. Следовательно, реализация проекта экономически целесообразна.

2 Проектирование сетевого кода

2.1 Выбор сетевой архитектуры

Глобально сетевой код выполняет две задачи: обеспечивает сетевое взаимодействие в процессе игры и реализует механизмы для соединения игроков в рамках игровой сессии. В данном разделе будет произведен выбор архитектуры сети в контексте первой задачи. Архитектура сети определяет основные элементы сети и определяет ее общую логическую организацию. Как правило, при разработке сетевой игры выбор стоит между архитектурой «клиент-сервер» и архитектурой «peer-to-peer» [2]. Выбор архитектуры зависит как от характера самой игры, так и от технических возможностей разработчиков. Рассмотрим преимущества и недостатки данных архитектур.

Архитектура клиент-сервер предполагает, что один экземпляр игры действует как выделенный сервер, в то время как остальные экземпляры игры считаются клиентами (рисунок 19) [2].

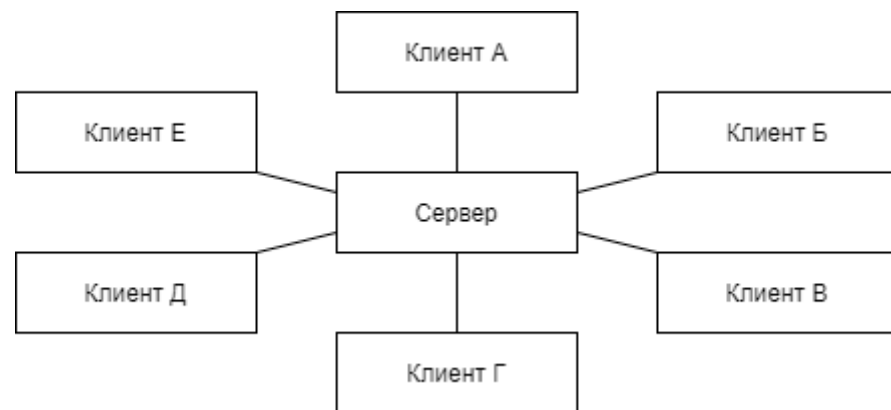


Рисунок 19 – Архитектура клиент-сервер

Чаще всего при использовании архитектуры клиент-сервер в играх сервер является уполномоченным (authoritative). Это означает, что только те игровые действия, которые выполнил сервер, считаются правильными [2]. Игрок может лишь только оповестить сервер о намерении выполнить то или иное игровое действие, после чего сервер принимает решение о возможности данного действия.

Такой подход приводит к тому, что действия игрока выполняются с задержкой, вклад в которую вносит время передачи/подтверждения (Round Trip Time, RTT) и время, необходимое на обработку клиентского действия. В быстрой сети эти задержки могут не ощущаться, однако на практике часто приходится использовать специальные приемы, скрывающие от игрока эти задержки [2].

Важно иметь в виду и особые требования к пропускной способности сервера, возникающие при такой архитектуре. Если каждому клиенту приходится поддерживать только одно соединение, с сервером, то серверу приходится поддерживать число соединений равное количеству клиентов. При росте числа клиентов линейно растут и требования к пропускной способности сервера [2].

В рамках архитектуры клиент-сервер можно выделить два основных вида игровых серверов: выделенный сервер (dedicated server) и прослушивающий (listen server).

Выделенный сервер занимается только обслуживанием состояния игры и обеспечивает взаимодействие клиентов. Он полностью отделен от клиентских процессов, выполняющих игру. Как правило, такие сервера используются в высокобюджетных играх [2].

Использование и содержание выделенного сервера сопряжено с большими затратами, однако взамен разработчики получают возможность снизить требования к вычислительным ресурсам клиентских машин. Кроме того, существенно сокращаются возможности для нечестной игры. Это связано с тем, что игровые сервера чаще всего являются уполномоченными и при этом выделенный сервер не находится под контролем ни одного из игроков. Следовательно, при выделенном сервере ни один игрок не может влиять на игровой мир напрямую [2].

Прослушивающий сервер, напротив, подразумевает активное участие сервера в игровом процессе. В этом случае один из клиентов берет на себя еще и роль сервера. Главным преимуществом такого подхода является

существенное снижение стоимости развертывания, так как арендовать или содержать выделенные сервера нет необходимости. Недостатком является то, что на клиента, выполняющего роль сервера, ложится дополнительная вычислительная и сетевая нагрузка [2].

Еще одним следствием использования прослушивающего сервера является то, что игрок, машина которого выполняет роль сервера, не испытывает сетевых задержек, а, следовательно, этот игрок получает преимущество перед остальными игроками. Помимо этого, если прослушивающий сервер является уполномоченным, игрок, чей компьютер был выбран в качестве сервера, теоретически, получает возможность использовать это обстоятельство для жульничества.

Архитектура peer-to-peer предполагает, что каждый игрок связан со всеми другими игроками, участвующими в игре (рисунок 20) [2].

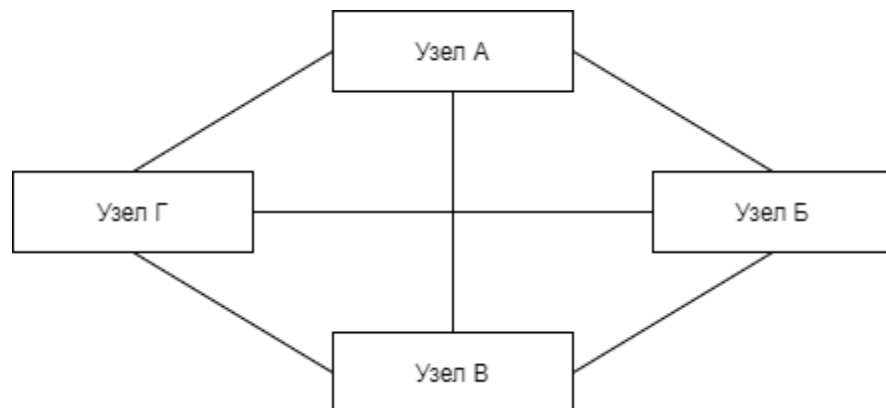


Рисунок 20 – Архитектура «peer-to-peer»

Такая организация сети позволяет реализовать взаимодействие узлов по-разному. Например, можно разделить ответственность за разные аспекты игры между разными узлами, однако реализация таких систем крайне сложна. Обычно каждый узел такой сети берет на себя ответственность за расчет состояния игрового мира. При этом каждый узел занимается пересылкой каждому узлу действий игрока. После получения этих действий узел применяет их к своему экземпляру игрового мира. Такую модель иногда называют моделью разделяемого ввода (input sharing model) [2].

Использование такой архитектуры ведет к тому, что уменьшаются накладные расходы на RTT, т.к. отсутствует посредник в виде сервера. При увеличении количества игроков сетевая нагрузка на каждый узел увеличится одинаково, а значит и требования к пропускной способности сети для всех игроков будут одинаковыми. Однако при использовании такой архитектуры накладные расходы на синхронизацию и поддержку согласованности состояния игры во всех узлах существенно возрастают. Потребность в согласовании состояния игрового мира дополнительно накладывает требование детерминированности на саму игру (один и тот же набор входных данных должен приводить к одному и тому же результату) [2].

Для разрабатываемого игрового приложения было принято решение использовать архитектуру «клиент-сервер» с уполномоченным прослушивающим сервером. Это позволит упростить синхронизацию, т.к. состояние игрового мира на машине, являющейся сервером, считается единственно правильным, и все клиенты вынуждены использовать именно его. Хост (игрок, компьютер которого используется в качестве игрового сервера) не будет испытывать дополнительных сетевых нагрузок, т.к. в игре участвует лишь 2 игрока. Кроме того, такой подход позволит без дополнительных трудозатрат использовать внутренние инструменты Unity для реализации игровой логики, в частности, физическую подсистему данного игрового фреймворка. Также это позволит значительно снизить требования к оборудованию, на котором будет развернут сервер игрового приложения, так как основной его задачей станет организация многопользовательских игровых сессий. Ответственность за расчет игровой логики возлагается на устройства игроков.

Проблема потенциального жульничества будет проигнорирована, т.к. разрабатываемая игра является кооперативной, а, следовательно, у игроков нет существенного повода для жульничества.

2.2 Выбор сетевого протокола

Не менее важным решением при разработке сетевой игры является выбор сетевого протокола. Как правило, выбор стоит между протоколами TCP и UDP, и зависит от характера игры.

Основным преимуществом протокола TCP является поддержка надежности. TCP гарантирует, что все отправленные данные будут доставлены до получателя точно в том порядке, в котором они были отправлены. Кроме того, протокол TCP реализует сложные механизмы, позволяющие избегать заторы в сети, регулируя скорость отправки пакетов.

Все это позволяет использовать TCP для реализации надежной передачи данных без дополнительных трудозатрат по обеспечению надежности. Однако такое преимущество протокола TCP оборачивается недостатком при передаче пакетов с данными, время доставки для которых критично.

Надежность TCP реализуется повторной передачей потерянных пакетов. Пока не будет принят очередной пакет, следующие за ними пакеты не могут быть приняты. Из-за такого подхода возможна ситуация, когда потеря низкоприоритетных данных ведет к задержке передачи высокоприоритетных данных. Даже в том случае, если в игре нет низкоприоритетных данных существует вероятность, что несколько потоков данных, требующих надежной передачи и упорядоченности, будут создавать задержки друг для друга. При этом для игрока в каждый момент времени может иметь наибольший приоритет конкретный поток данных, а потому надежная передача пакета из одного потока данных в ущерб оперативной передаче пакета из другого потока данных может значительно ухудшить игровой опыт. И даже если существует только один поток данных, безальтернативная надежность TCP может создать проблемы. Если очередной пакет при передаче был потерян, может оказаться так, что информация в этом пакете успеет устареть. В таком случае часто бывает эффективнее проигнорировать потерю пакета и отправить актуальные данные, однако TCP не предоставляет такой возможности.

Механизмы обнаружения и устранения заторов, реализуемые TCP, с одной стороны позволяют уменьшить потерю пакетов, с другой, могут привести к замедлению передачи пакетов. Поэтому, при использовании протокола TCP в игровых приложениях алгоритмы устранения заторов, как правило, отключаются.

Протокол UDP, напротив, не реализует никаких механизмов надежной передачи. Данный протокол не гарантирует ни доставку пакетов, ни их порядок. UDP только позволяет отправить пакет по заданному ip и порту, а реализация надежных механизмов передачи становится ответственностью разработчика. Единственным механизмом надежности, реализованным в UDP, является шестнадцатитбитная контрольная сумма, которая все равно не может обеспечить достаточного уровня защиты.

Однако взамен UDP позволяет реализовать на своей основе любую требуемую систему передачи данных. Можно организовать как надежную передачу, так и ненадежную, можно принимать решения о повторной передаче данных исходя из приоритета и актуальности информации, можно гибко управлять несколькими потоками надежной передачи данных и т.д. Все это делает протокол UDP оптимальным выбором для большинства игр [2, 18].

Так как в рамках выпускной квалификационной работы необходимо реализовать экшен-игру, своевременная передача данных становится критически важной. Следовательно, необходимо реализовать собственную систему передачи данных на основе протокола UDP.

2.3 Проектирование матчмейкера

Ранее для реализации сетевого взаимодействия между игроками была выбрана архитектура «клиент-сервер» с уполномоченным прослушивающим сервером, подразумевающая, что один из игроков будет выбран в качестве хоста. Однако при такой реализации все еще существует необходимость в центральном сервере-матчмейкере, ответственном за создание игровых сессий

и соединение игроков друг с другом в рамках этой сессии. Произведем проектирования сервера-матчмейкера.

2.3.1 Логическое проектирование базы данных

Произведем логическое проектирование базы данных (рисунок 21).

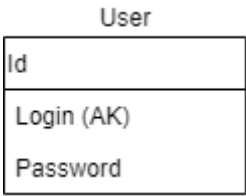


Рисунок 21 – Логическая модель базы данных

В таблице 5 представлено описание сущности User.

Таблица 5 – Описание сущности «User»

Наименование	Описание	Тип данных	Размерность	Область допустимых значений	Возможность значения null	Роль	Пример
Id	Номер	Целый.	10	1-2 147 483 648	Нет	РК	1
Login	Логин пользователя	Символьный	20	a-zA-Z0-9_	Нет	АК	Ivan
Password	Пароль пользователя	Символьный	20	a-zA-Z0-9_	Нет		12345678

Сущность User содержит необходимую информацию для реализации регистрации и входа в систему.

2.3.2 Проектирование API

Взаимодействие с сервером-матчмейкером не накладывает жестких ограничений на скорость передачи данных. В то же время надежность передачи данных в данном случае имеет большое значение. В таких условиях имеет смысл использовать надежный протокол передачи данных, такой как TCP или HTTP.

Для реализации сервера-матчмейкера было решено реализовать web-сервис, взаимодействие с которым реализуется по протоколу HTTP. В качестве формата передаваемых данных был выбран JSON.

Для реализации сервера-матчмейкера будет использован фреймворк ASP.NET Core. Средствами данного фреймворка необходимо реализовать веб-API, позволяющее пользователям регистрироваться и входить в систему, а также участвовать в игровых матчах. Применение фреймворка ASP.NET Core подразумевает использование архитектуры MVC. На рисунке 22 продемонстрирована общая структура разрабатываемого веб-API [19].

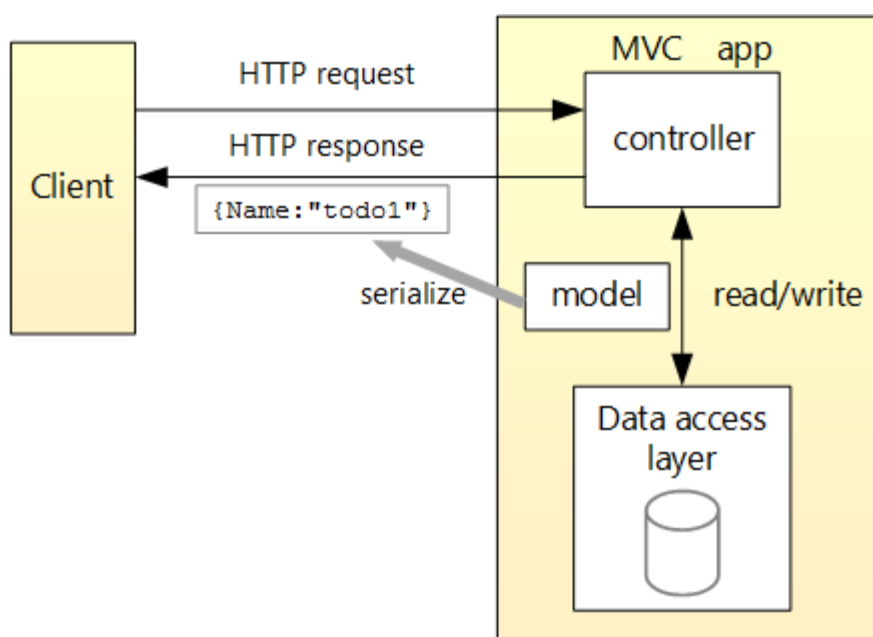


Рисунок 22 – Структура веб-API

ASP.NET Core имеет встроенную поддержку механизма внедрения зависимостей. Это позволяет без труда реализовывать и внедрять в контроллеры сервисы соблюдая при этом принцип инверсии зависимости [20]. Было принято решение использовать данный механизм для реализации логики создания игровых матчей. Логiku создания матчей необходимо реализовать в виде сервиса, доступ к которому будет реализован через соответствующий контроллер. Упрощенная схема взаимодействия клиента с сервисом создания матчей продемонстрирована на рисунке 23.

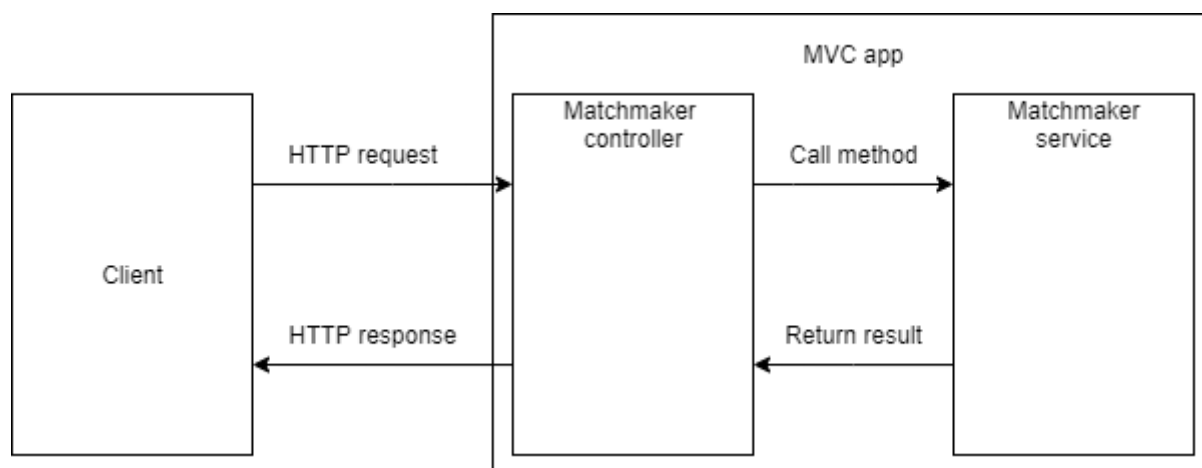


Рисунок 23 – Упрощенная схема взаимодействия с сервисом создания матчей.

На рисунке видно, что клиент обращается к контроллеру матчмейкера, после чего клиентские запросы перенаправляются сервису создания матчей с помощью вызова соответствующих методов.

2.4 Проектирование клиентской части приложения

Клиентская часть в данном случае представляет собой игровое приложение. Разработка игровых приложений имеет свои особенности, связанные с тем, что игры особенно подвержены изменениям в процессе разработки. Крайне часто происходит так, что итоговый игровой продукт полностью отличается от изначальной задумки. Это связано с тем, что очень сложно наперед предугадать, какие игровые механики, которые могут казаться интересными на бумаге, окажутся интересными на практике, что приводит к частым изменениям. Поэтому критически важно заложить возможности для изменения игровых механик изначально, иначе разрабатываемое игровое приложение рискует стать трудноподдерживаемым.

Еще одной важной особенностью игровых приложений является то, что в основе практически любой игры лежит игровой цикл. Игровой цикл вызывается, обычно, 60 или 30 раз в секунду и обновляет состояние игрового мира. Каждая итерация цикла должна выполнить всю актуальную игровую логику. Такая особенность может привести к тому, что код, ответственный за

обработку игровой логики начнет сосредотачивать в теле игрового цикла, что приведет к его неконтролируемому росту. В итоге игровой цикл рискует стать настолько сложным, что поддержка игрового приложения станет практически невозможной.

Чтобы избежать описанные выше проблемы было решено использовать архитектурный шаблон ECS (Entity Component System). ECS – один из универсальных и хорошо зарекомендовавших себя архитектурных шаблонов для разработки игровых приложений. Данный подход подразумевает использование композиции вместо наследования, что значительно повышает гибкость и расширяемость приложения.

ECS предполагает разделение приложения на три основных части: сущности, компоненты и системы. Рассмотрим их подробнее.

Сущности (Entity), представляют любые объекты игрового мира и являются объектами-контейнерами, содержащими компоненты. Они лишены свойств или логики.

Компоненты (Component) представляют собой атомарные блоки данных, определяющие те или иные свойства сущностей.

Системы (System) в свою очередь реализуют тот или иной аспект игровой логики.

Идея архитектурного шаблона ECS заключается в том, чтобы разделить данные и игровую логику. Классический ООП подход предполагает построение иерархий классов, которые при неудачном проектировании рискуют разрастись и углубиться настолько, что поддержка такого решения станет практически невозможной. ECS позволяет избежать этого, предлагая использовать вместо наследования композицию. Для того, чтобы определить новый объект в рамках игрового мира достаточно создать новую сущность и наделить ее теми компонентами, которые опишут все необходимые свойства данного объекта. При этом игровая логика полностью отделена от сущностей и компонентов и локализована внутри систем. То, какими системами будет обрабатываться сущность полностью определяется ее компонентами. Так,

например, для системы, рассчитывающей перемещение сущности, необходимо наличие у сущности компонентов, определяющих направление движения и модуль скорости. Следовательно, все сущности, обладающие данными компонентами, будут обработаны в такой системе, в то время как все остальные сущности будут проигнорированы [21 – 22].

Такое разделение игровой логики и данных приводит к тому, что приложение становится гибким и масштабируемым. С одной стороны, для создания новых игровых объектов достаточно определить недостающие компоненты (или скомбинировать уже имеющиеся компоненты определенным образом), с другой, для добавления, удаления и изменения игровой логики достаточно добавить, удалить или изменить соответствующую систему. При этом такие изменения не коснутся других частей кода. Кроме того, такая локализация игровой логики упрощает тестирование.

Игровой цикл при реализации шаблона ECS имеет простую структуру. Поскольку вся игровая логика определена в системах, игровому циклу достаточно вызвать все активные системы игрового приложения в нужной последовательности.

К недостаткам ECS можно отнести увеличение количества кода и зависимость от порядка вызова игровых систем, однако эти недостатки, как правило, являются несущественными, особенно в контексте небольших проектов [21].

С точки зрения сетевого кода использование ECS удобно потому, что состояние игрового мира в каждый момент времени можно представить в виде набора всех существующих сущностей и их компонентов. Это позволяет значительно упростить задачу сериализации и десериализации игрового мира. Было принято решение реализовать собственный ECS фреймворк, который бы обладал простой внутренней структурой. При реализации были использованы идеи ECS фреймворка Ash для языка Actioscrip [23]. От использования уже существующего порта данного фреймворка на язык C# Ash.NET было решено

отказаться, так как работы по портированию были прекращены в 2012 году, и так и не были полностью закончены. Кроме того, Ash.NET является точной копией исходного фреймворка, несмотря на то, что возможности языка C# могли бы принести значительные улучшения [24]. На рисунке 24 представлена диаграмма классов, демонстрирующая, общую структуру ECS фреймворка.

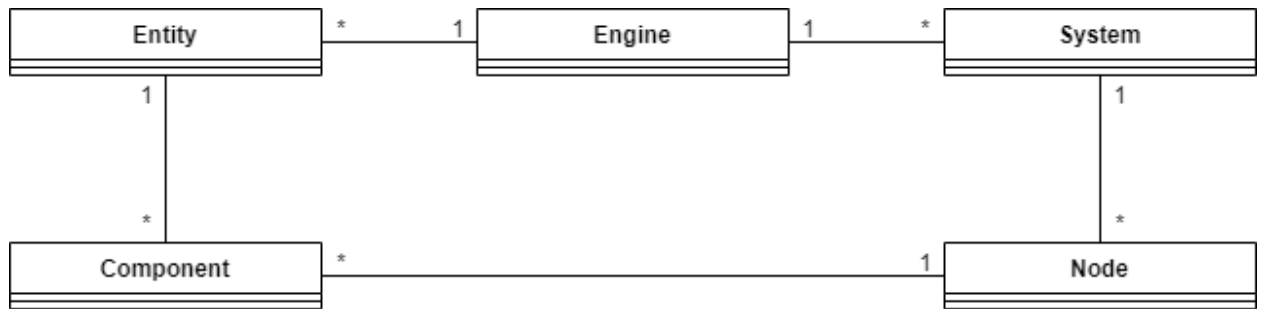


Рисунок 24 – Диаграмма классов, демонстрирующая общую структура ECS фреймворка

На диаграмме видно, что помимо сущностей, компонентов и систем в фреймворке имеются классы-узлы (Node). Их задача заключается в том, чтобы определить, какие компоненты нужны для каждой конкретной системы.

2.5 Проектирование алгоритмов

2.5.1 Проектирование алгоритма для сервиса создания матчей

На рисунке 25 изображена диаграмма деятельности, демонстрирующая алгоритм работы сервиса для создания матчей.

Из диаграммы видно, что сервис выполняет две параллельных работы. Он отслеживает игроков в очереди, которые давно не совершали запросы, и удаляет их, так как такие игроки, скорее всего, аварийно прекратили поиск матча, не оповестив об этом сервис. В то же время сервис отслеживает количество игроков в очереди, и как только в ней набирается достаточное количество игроков, сервис создает для них матч.

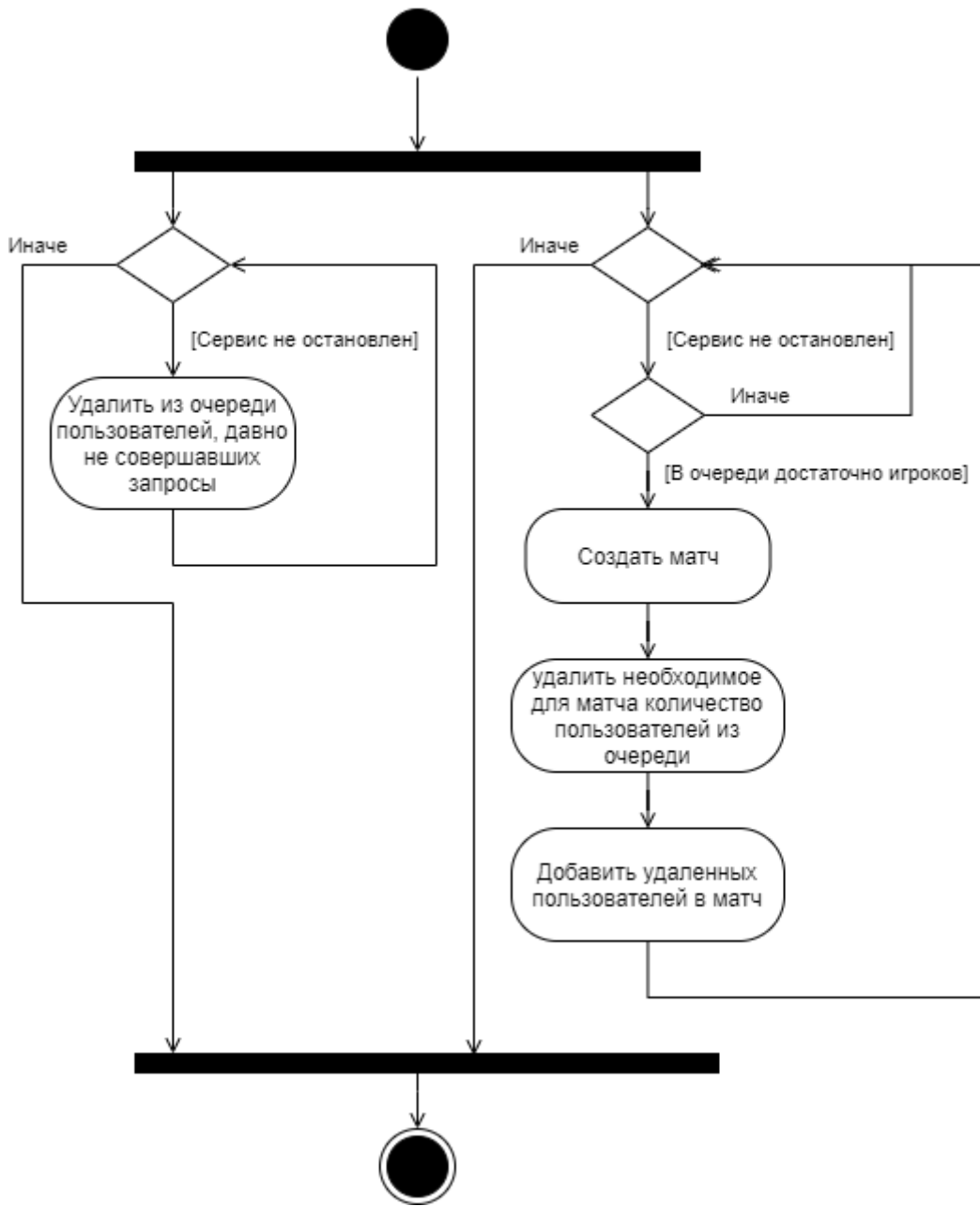


Рисунок 25 – Диаграмма деятельности, демонстрирующая работу сервиса для создания матчей

Имеет смысл реализовать сервис для матчмейкинга таким образом, чтобы он не зависел от вида создаваемых матчей. В таком случае данный алгоритм можно будет повторно применять для создания матчей с разными игровыми режимами в будущем.

2.5.2 Проектирование алгоритма для игрового матча

Так как ответственность за обработку игровой логики ложится на одного из игроков, на стороне сервера-матчмейкера необходимо только узнать

публичные и приватные конечные точки каждого игрока, а также идентификатор матча, и отправить эти данные участникам игры, чтобы они смогли соединиться друг с другом. Необходимость в публичных и приватных конечных точках и в идентификаторе матча будет раскрыта в следующем разделе. Сущность, ответственную за получение и передачу этих данных участникам игры назовем матчем. На рисунке 26 представлена диаграмма состояний, демонстрирующая работу матча.

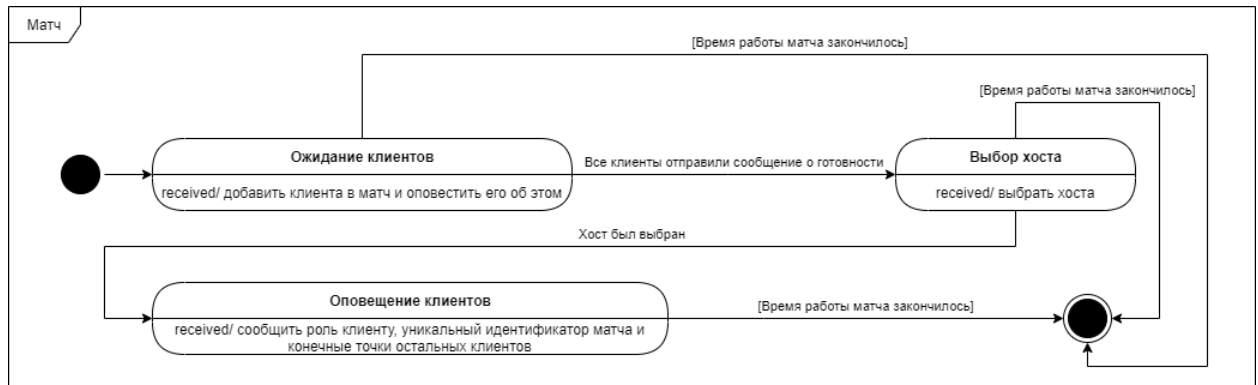


Рисунок 26 – Диаграмма состояний матча

Процесс соединения клиентов можно поделить на 3 основных этапа. Во время первого этапа матч ожидает сообщений от клиентов-участников игры. Когда клиент обращается к матчу, его конечные точки сохраняются. После того, как все клиенты совершат свое первое обращение к матчу, начинается второй этап. Во время второго этапа выбирается хост (игрок, компьютер которого будет выполнять как роль клиента, так и роль сервера). После того как хост будет выбран произойдет переход к третьему этапу, во время которого всем клиентам сообщается их роль (хост или клиент), необходимые им конечные точки и идентификатор матча. Хосту сообщаются конечные точки всех остальных игроков, в то время как клиентам сообщается только конечная точка хоста. Получив эти данные, клиенты могут начать соединение.

2.5.3 Проектирование алгоритма UDP Hole Punching

После того, как игроки получили свои роли, идентификатор матча и конечные точки других игроков, необходимо обеспечить их непосредственное

взаимодействие. В общем случае клиенты не могут общаться напрямую, используя протокол UDP или TCP, из-за механизма NAT (Network Address Translation). Чтобы обойти это ограничение для протокола UDP можно воспользоваться алгоритмом UDP Hole Punching. Идея этого алгоритма заключается в использовании сервера с известным глобальным ip для обеспечения соединения клиентов непосредственно друг с другом. Рассмотрим работу алгоритма UDP Hole Punching на примере двух клиентов. Будет рассмотрен наиболее распространенный случай, когда клиенты находятся за разными NAT.

Пусть, необходимо обеспечить взаимодействие клиента А и клиента В. Роль сервера, обеспечивающего соединение клиентов, выполняет сервер S. Чтобы инициировать процесс соединения, клиенты А и В отправляют соответствующие запросы на сервер S, сообщая ему при этом свои приватные конечные точки. Сервер S получает приватные конечные точки клиентов А и В и определяет их публичные конечные точки, используя для этого заголовки дейтаграмм. Схема передачи сообщений до этапа соединения представлена на рисунке 27.

После того, как сервер S узнал приватные и публичные конечные точки клиентов соединение между клиентами может быть установлено. Клиент А запрашивает конечные точки клиента В. В свою очередь клиент В запрашивает конечные точки клиента А. Получив конечные точки клиенты начинают слать друг другу дейтаграммы, используя для этого как приватную, так и публичную конечную точку. Клиенты должны использовать тот же порт, что они использовали для обращения к серверу S. В таком случае NAT, за которыми находятся клиенты, использует порт, что был использован для обращения к серверу S, для любых новых исходящих сессий. Однако существуют ситуации, когда NAT определяет новый порт для каждой новой сессии. По такому принципу работает симметричный NAT. В таком случае алгоритм UDP Hole Punching не сработает.

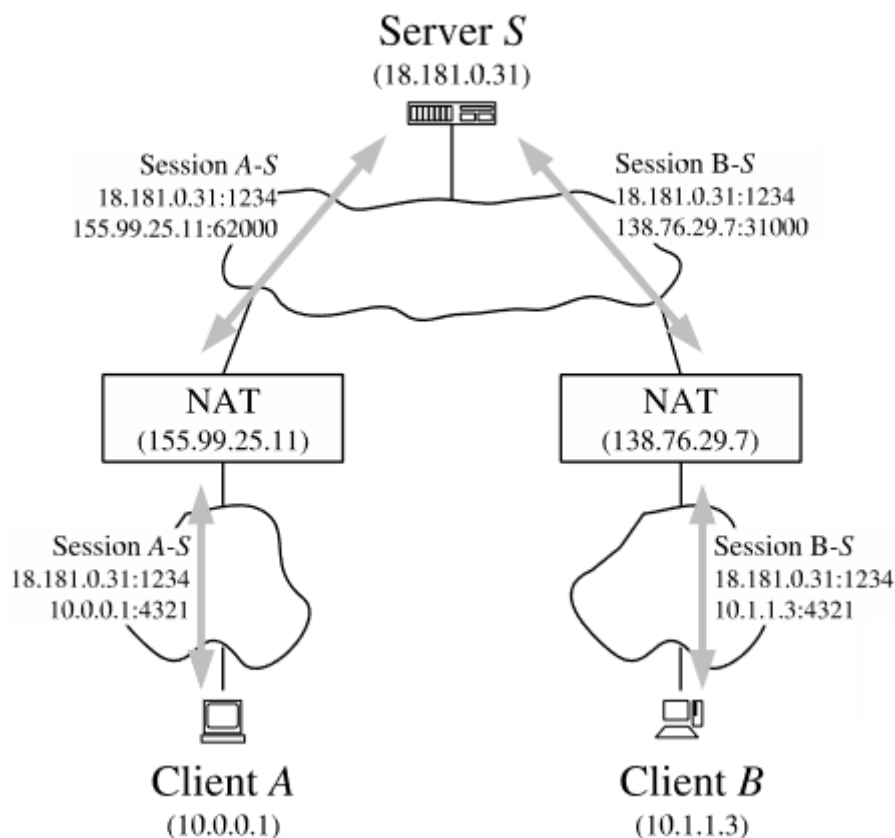


Рисунок 27 – Схема передачи сообщений до соединения

Обращаясь друг к другу, клиенты создадут в своих NAT соответствующие записи, которые позволят им принимать друг от друга сообщения. Пусть клиент А обратится к клиенту В первым. Отправляя дейтаграмму клиенту В клиент А создаст в своем NAT запись, в которой в качестве адреса получателя будет записана публичная конечная точка клиента В. Так как NAT клиента В не ожидает сообщений от клиента А, дейтаграмма будет проигнорирована. Однако дейтаграмма, которую после отправит клиент В клиенту А, уже сможет достичь получателя, так как клиент А создал необходимую запись в своем NAT первой «холостой» дейтаграммой. Процесс обмена дейтаграммами между клиентами с целью установить соединение продемонстрирован на рисунке 28.

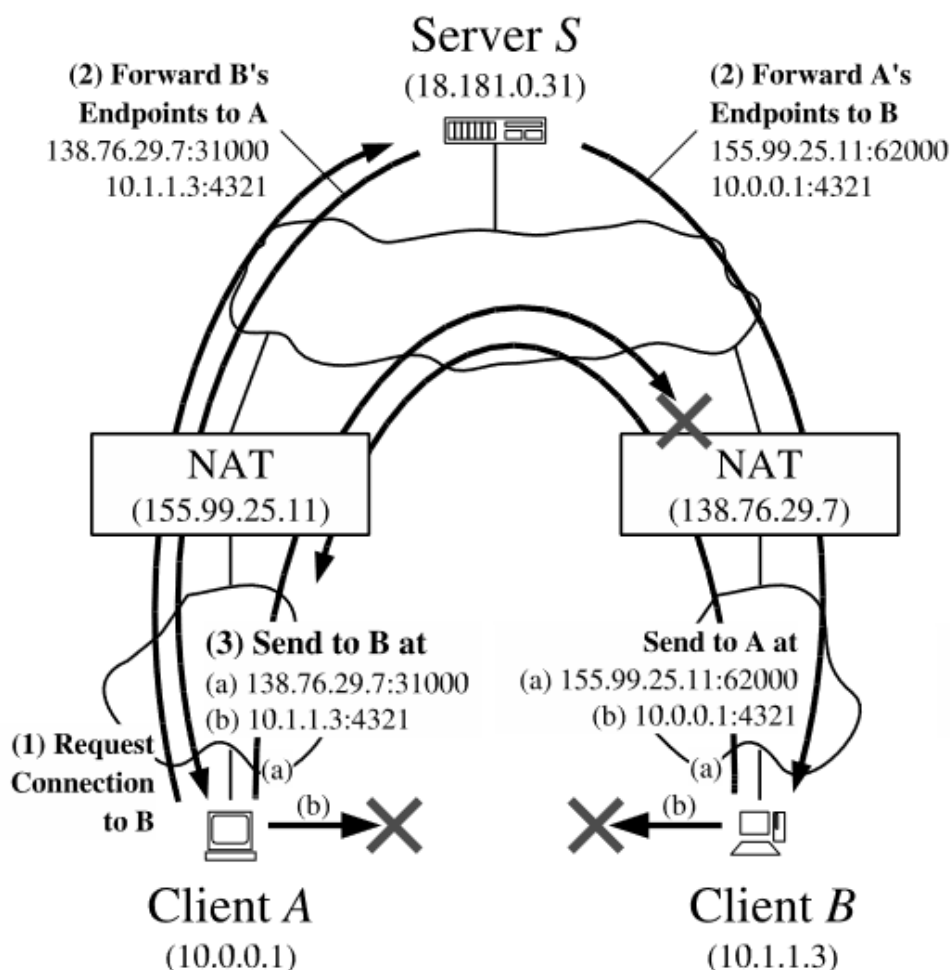


Рисунок 28 – Обмен дейтаграммами с целью установить соединение

Обратим внимание, что перед процессом соединения клиентам сообщается помимо публичной конечной точки еще и приватная. Она необходима в том случае, если оба клиента окажутся за одним и тем же NAT. В таком случае общение между клиентами будет происходить при помощи приватных конечных точек.

Из-за того, что клиенты вынуждены опрашивать и публичную, и приватную конечные точки, существует вероятность коллизии приватной конечной точки клиента, с которым происходит соединение, и приватной конечной точки другого компьютера в локальной сети. Следует реализовать механизм защиты, предотвращающий возможные негативные последствия от такого совпадения. В рамках разрабатываемого приложения для этих целей

используется идентификатор матча, позволяющий однозначно определить, пришло ли сообщение от участника данного матча.

После того как клиенты обменялись дейтаграммами, соединение между ними установлено. Процесс общения клиентов после соединения представлен на рисунке 29 [25].

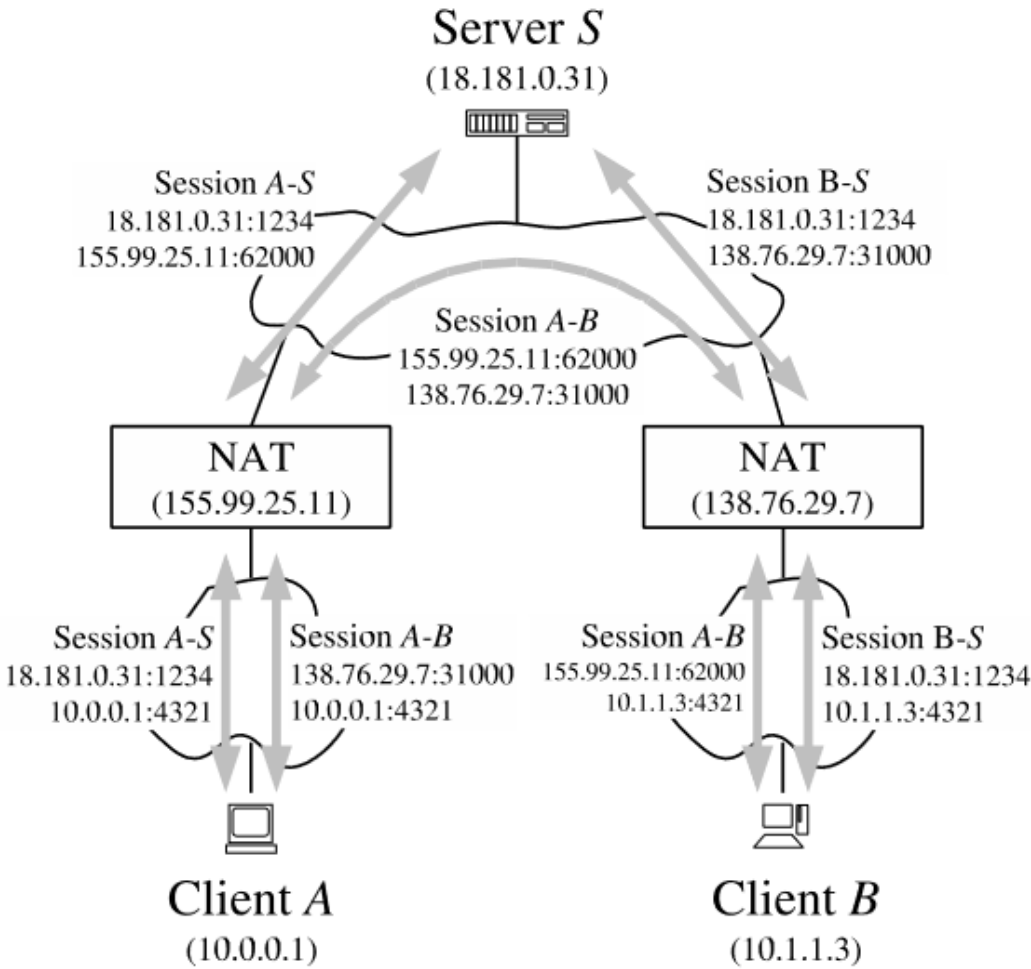


Рисунок 29 – Схема передачи сообщений после соединения

В контексте разрабатываемого приложения задачу получения конечных точек и оповещения клиентов решает матч, алгоритм которого был описан ранее. Упрощенный алгоритм, реализующий соединение клиентов между собой продемонстрирован на диаграмме деятельности (рисунке 30).

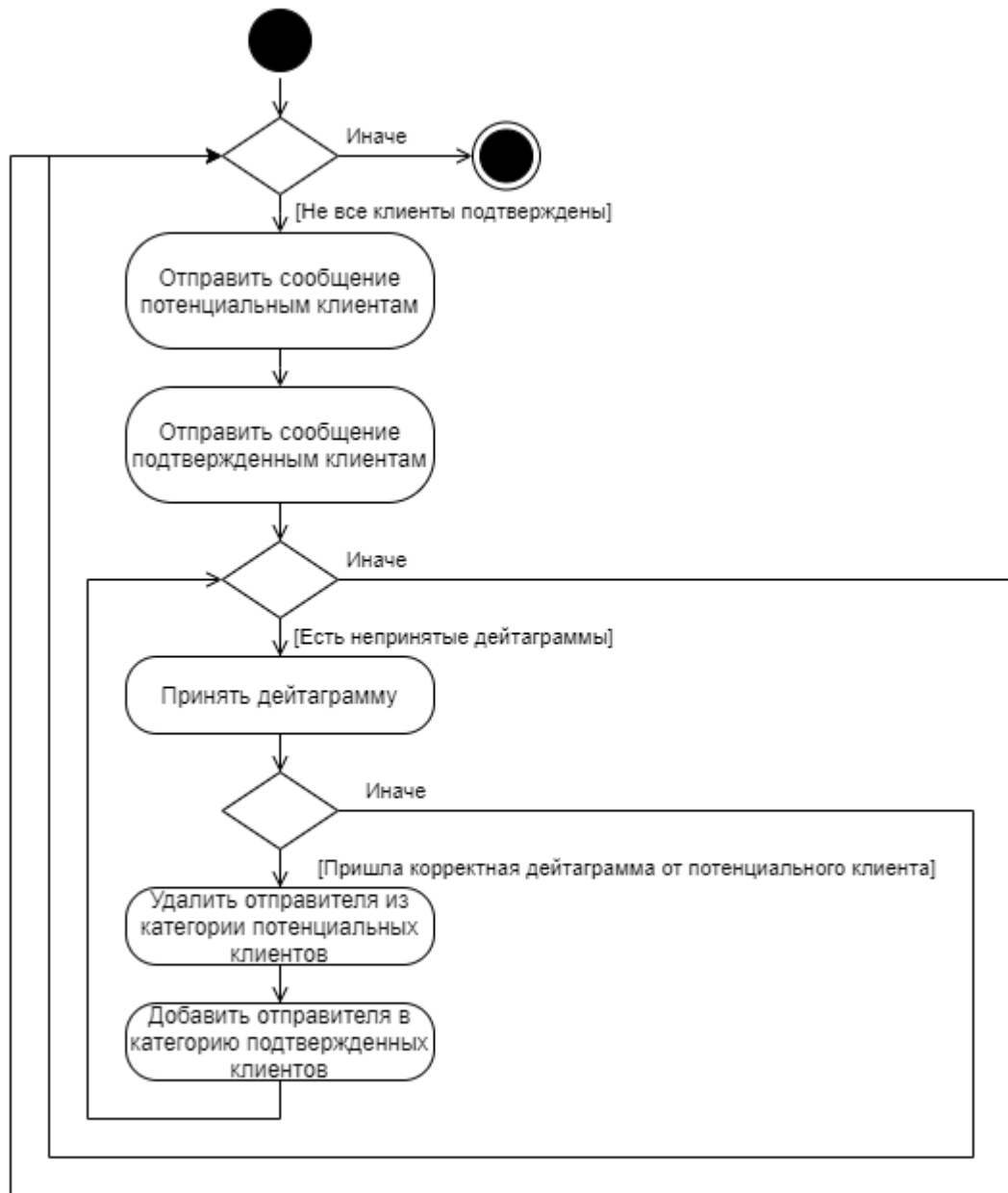


Рисунок 30 – Диаграмма деятельности, демонстрирующая алгоритм соединения клиентов

На рисунке видно, что в процессе соединения выделяются две категории пользователей: потенциальные и подтвержденные. Потенциальными считаются те пользователи, от которых пока не было получено ни одной дейтаграммы. Следовательно, для таких клиентов нельзя с уверенностью сказать, готовы ли они к обмену данными. После того, как первое сообщение от клиента будет получено, можно сделать вывод, что соединение с данным клиентом установлено.

2.5.4 Проектирование алгоритма обмена данными

После того, как соединение было установлено, между игроками может быть реализован обмен данными. Игрок может выполнять либо роль клиента, либо роль хоста. То какие данные принимает и отправляет игрок зависит от его роли. На рисунке 31 представлена диаграмма деятельности, демонстрирующая общую схему приема и передачи данных на стороне хоста.

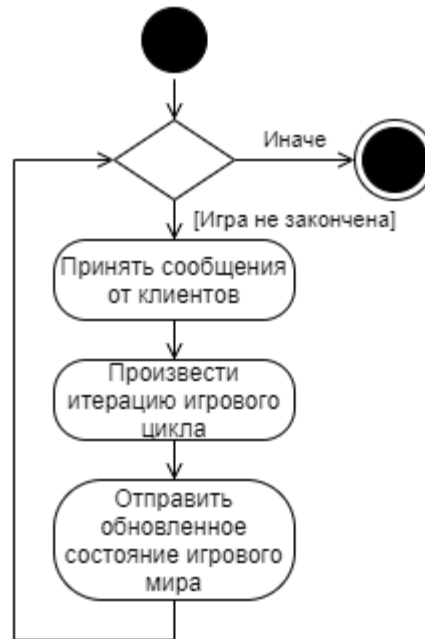


Рисунок 31 – Общая схема приема и передачи данных на стороне хоста

На диаграмме видно, что перед каждой итерацией игрового цикла хост принимает сообщения от клиентов. Так как в рамках разрабатываемого игрового приложения необходимо реализовать уполномоченный сервер, клиентские сообщения будут содержать только пользовательский ввод. После, хост производит итерацию игрового цикла, во время которой ввод всех клиентов и хоста применяется к игровому миру. В конце итерации игровой мир переходит в новое состояние, которое хост сообщает клиентам.

Диаграмма деятельности, демонстрирующая общую схему приема и передачи данных на стороне клиента, представлена на рисунке 32.

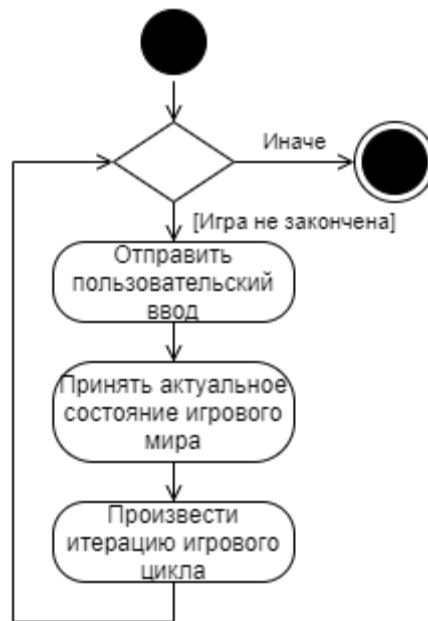


Рисунок 32 – Общая схема приема и передачи данных на стороне клиента

Перед каждой новой итерацией игрового цикла, клиент отправляет свой ввод хосту. После, клиент принимает актуальное состояние игрового мира и применяет его к своему игровому миру. Далее клиент производит итерацию игрового цикла.

Из рассмотренных диаграмм видно, что хост передает клиентам состояние всего мира целиком. Такое решение было принято потому, что такой подход с одной стороны прост в реализации, с другой, хорошо подходит динамичным играм. Однако при этом значительно увеличивается объем трафика. Следовательно, для стабильной работы приложения могут потребоваться техники оптимизации трафика.

2.6 Проектирование формата для обмена данными

Из-за того, что хост вынужден на каждой итерации игрового цикла передавать состояние всего мира целиком, оптимизация трафика может стать важной задачей. Одно из ограничений, которое необходимо учитывать при реализации сетевого кода, является размер MTU (Maximum Transmission Unit). По умолчанию он равен 1500 байт [2]. Следовательно, если передаваемый пакет превысит данное значение, он будет разделен на несколько пакетов.

Такое поведение нежелательно, т.к. в результате значительно усложняется процесс обмена данными и, что более критично, замедляется сетевое взаимодействие.

Несмотря на то, что в рамках разрабатываемой игры для описания игрового мира не требуется много данных, тем не менее было решено реализовать свой формат для обмена данными, организовав тем самым основу для дальнейшего расширения игры.

Формат может быть описан при помощи следующей грамматики G:

$N = \{\text{ДАнные}, \text{ТОКЕН}, \text{СИМВОЛЫ}, \text{СИМВОЛ}\}$
 $E = \{a, b, c, \dots A, B, C, \dots 0, 1, 2, \dots\}$

$P =$
 $\{$
 ДАнные $\rightarrow$ ТОКЕН ДАнные
 ДАнные $\rightarrow e$
 ТОКЕН $\rightarrow$ "СИМВОЛЫ"
 СИМВОЛЫ $\rightarrow$ СИМВОЛ СИМВОЛЫ
 СИМВОЛЫ $\rightarrow e$
 СИМВОЛ $\rightarrow a|b|c\dots A|B|C\dots 0|1|2\dots$
 $\}$

$S = \text{ДАнные}$

Данная грамматика описывает простой формальный язык, все предложения которого представляют собой последовательности токенов. Токены могут состоять из любых символов. Детерминированный конечный автомат (ДКА) для разбора предложений грамматики G представлен на диаграмме состояний (рисунок 33).

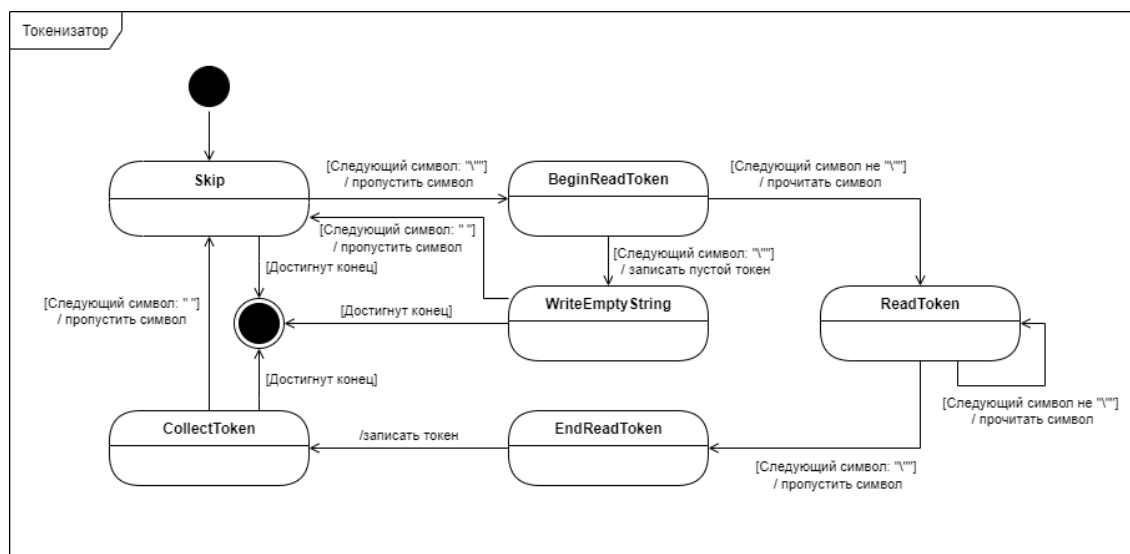


Рисунок 33 – Диаграмма состояний, демонстрирующая работу ДКА для разбора грамматики G

Результатом работы автомата является последовательность токенов, пригодная для дальнейшего анализа и интерпретации.

Так как последовательность токенов сама по себе не обладает семантикой, ее необходимо обеспечить на более высоком уровне. Для этого было принято решение сериализовывать данные в строгом порядке. Для того, чтобы продемонстрировать структуру сообщения, рассмотрим пример. В листинге 1 представлены компоненты, которые необходимо сериализовать.

Листинг 1 – Сериализуемые компоненты

```
public class IdComponent : ComponentBase
{
    public uint Id { get; set; }
}

public class VelocityComponent : ComponentBase
{
    public float X { get; set; }
    public float Y { get; set; }
}

public class HealthComponent : ComponentBase
{
    public uint CurrentHealth { get; set; }
    public uint MaxHealth { get; set; }
}

public class IsAliveComponent : ComponentBase { }
```

Поставим в соответствие каждому компоненту уникальный идентификатор (таблица 6).

Таблица 6 – Соответствие компонентов и идентификаторов

Компонент	Идентификатор
IdComponent	0
VelocityComponent	1
HealthComponent	2
IsAliveComponent	3

Рассмотрим пример сообщения и продемонстрируем его структуру (рисунок 34).

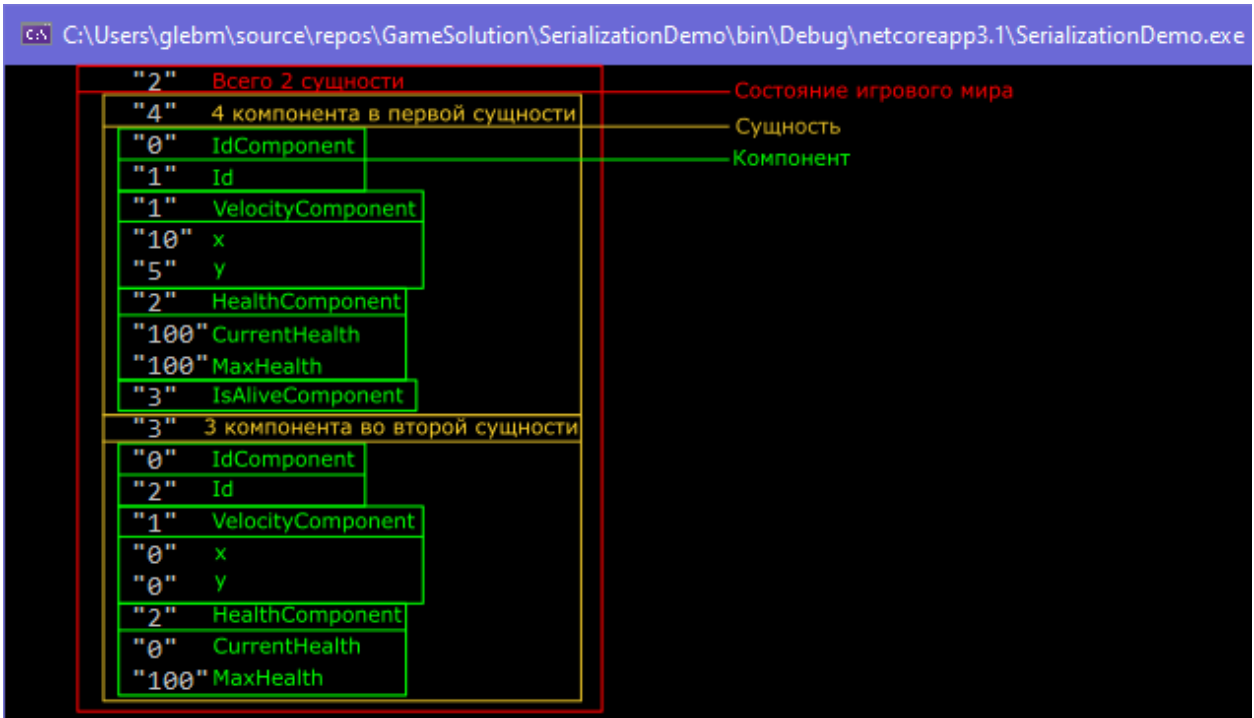


Рисунок 34 – Демонстрация структуры сообщения

Первый токен последовательности определяет количество сериализованных сущностей. Второй токен определяет число компонентов в первой сущности. Третий токен определяет идентификатор первого компонента сущности. Последующие токены, вплоть до следующего идентификатора компонента, представляют собой сериализованные данные первого компонента и т.д. Такой подход к сериализации данных значительно более экономичен, чем использование таких форматов как xml или json. При этом в перспективе такой способ сериализации можно будет сделать еще более экономичным реализовав сериализацию непосредственно в байты.

3 Реализация сетевого кода

3.1 Реализация матчмейкера

3.1.1 Реализация базы данных

Для работы с базой данных было решено использовать ORM Entity Framework Core. Для создания базы данных был использован подход Code First. При таком подходе сущности базы данных описываются при помощи классов-моделей, а связи между сущностями описываются в специальном классе-контексте базы данных [26]. На рисунке 35 продемонстрирована диаграмма классов, описывающая классы-модели приложения.

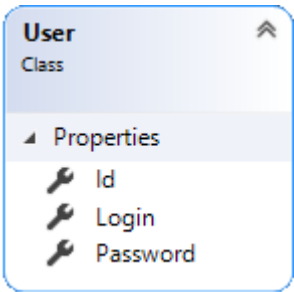


Рисунок 35 – Диаграмма классов-моделей

После описания базы данных, необходимо произвести миграцию, чтобы ее создать. На рисунке 36 представлена схема базы данных, сгенерированная с помощью инструмента DBeaver, демонстрирующая базу данных, полученную после миграции.

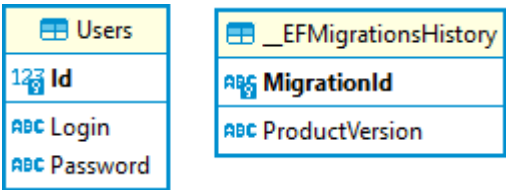


Рисунок 36 – Схема базы данных

Помимо таблицы User в базе данных имеется таблица, сгенерированная Entity Framework. Она содержит информацию о миграциях.

3.1.2 Реализация API

Архитектура ASP.NET Core MVC предполагает, что ответственность за обработку клиентских запросов лежит на классах-контроллерах. При получении запроса, система выбирает нужный контроллер и передает ему данные запроса. Фреймворк ASP.NET Core позволяет ассоциировать запросы с методами контроллера при помощи атрибутов. В таблице 7 представлена спецификация контроллеров, реализованных в приложении.

Таблица 7 – Спецификация контроллеров

Метод	Описание	URI	Контроллер
Enqueue(ClientEndPoint privateEndPoint)	Ставит клиента в очередь на матч.	PUT api/matchmaker/queue	MatchmakerController
GetStatus()	Возвращает статус игрока.	GET api/matchmaker/status	
GetMatch()	Возвращает порт, на котором запущен матч, в котором участвует игрок.	GET api/matchmaker/match	
Remove()	Удаляет игрока из очереди.	DELETE api/matchmaker/queue	
Register(UserViewModel model)	Регистрирует нового пользователя в системе.	POST api/account/registration	AccountController
CheckAuthorization(UserViewModel model)	Проверяет, вошел ли пользователь в систему.	GET api/account/authorization	
Authorize(UserViewModel model)	Реализует вход пользователя в систему.	POST api/account/authorization	

Диаграмма компонентов, демонстрирующая взаимосвязь модулей серверной части приложения продемонстрирована на рисунке 37.

В таблице 8 представлено описание компонентов.

Таблица 8 – Описание компонентов серверной части приложения

Компонент	Описание
Matchmaker.exe	Исполняемый файл серверной части приложения.
Matches.dll	Библиотека, содержащая реализации игровых матчей.
Models.dll	Библиотека, содержащая классы-модели.
Context.dll	Библиотека, содержащая контекст базы данных.
Network.dll	Библиотека, содержащая вспомогательные классы для организации сетевого взаимодействия.

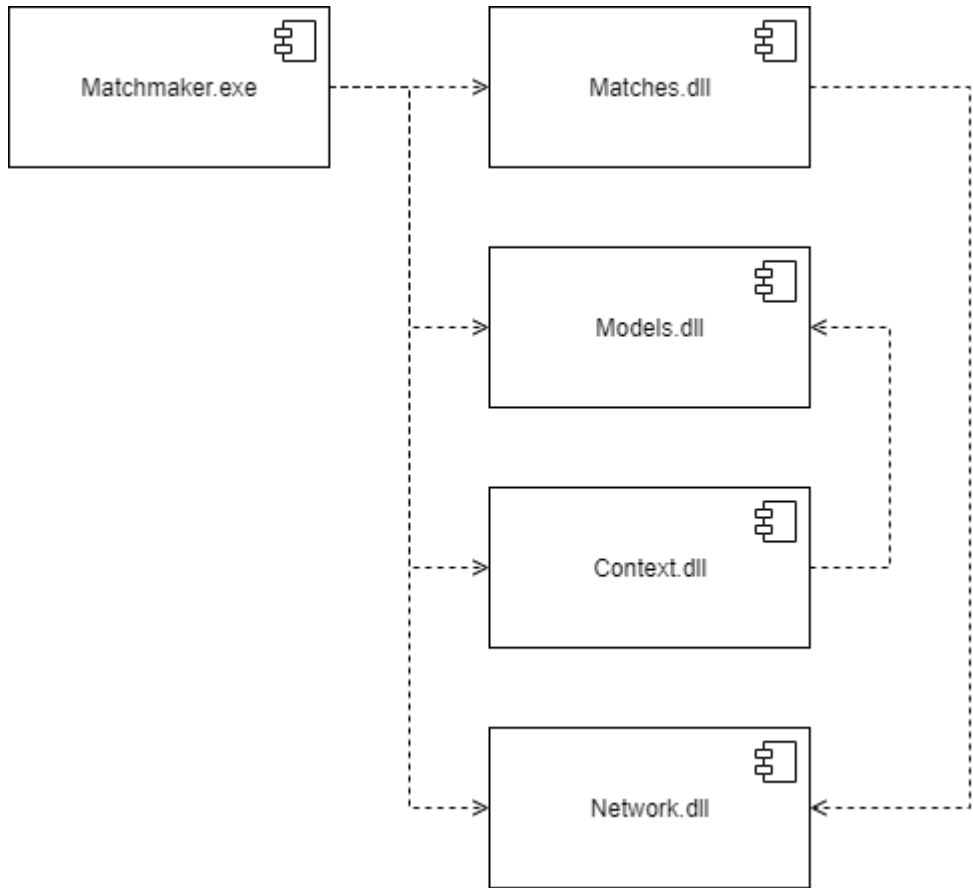


Рисунок 37 – Диаграмма компонентов серверной части приложения

Исходный код основных классов веб-API представлен в приложении Б.

3.2 Реализация клиентской части приложения

При разработке сетевой игры критически важно учитывать особенности сетевых игр в целом, и разрабатываемого проекта в частности. Поэтому разработка клиентской части приложения велась в тесном сотрудничестве с членом команды, ответственным за геймдизайн. Для организации совместной работы над проектом была использована система контроля версий git. Для онлайн хостинга репозитория, чтобы с проектом можно было взаимодействовать удалено, использовался GitHub. Для того, чтобы проект оставался управляемым, а командное взаимодействие эффективным, было принято решение следовать модели рабочего процесса Gitflow [27].

Клиентское приложение разрабатывалось с помощью игрового фреймворка Unity. Несмотря на то, что Unity «из коробки» поддерживает

компонентный подход, реализация, представленная в Unity, не является полноценным ECS, так как компоненты Unity объединяют внутри себя как данные, так и логику. Тем не менее средства Unity позволяют без труда внедрить собственную реализацию ECS. Достаточно разнести игровую логику, которая обычно реализуется в классах, наследующихся от базового класса `MonoBehaviour`, по игровым системам и создать компоненты-обертки над компонентами Unity.

Для того чтобы упростить разработку части программы, ответственной за взаимодействие с матчмейкером, матчем и за соединение клиентов, было принято решение выделить данный функционал в отдельные сущности (далее коннекторы). Для матчмейкера, матча и алгоритма UDP Hole Punching были реализованы собственные коннекторы, что позволило с одной стороны инкапсулировать процесс сетевого взаимодействия с сервером и процесс соединения клиентов, с другой, позволило разрабатывать данные компоненты за пределами проекта Unity, что в свою очередь дало больше возможностей для тестирования. Исходный код коннекторов представлен в приложении В.

В процессе игры хосту и клиенту необходимо обмениваться данными. Для того, чтобы посылка и прием не отнимало время (которое может быть существенным) на каждой итерации игрового цикла, сетевое взаимодействие было реализовано в отдельном потоке. Специально для этого были разработаны классы-прокси для клиента и для хоста, принимающие входящие дейтаграммы, и предоставляющие эти данные потоку, ответственному за обработку игровой логики, по запросу. Аналогичным образом данные классы работают и на передачу, принимая из основного потока данные и отправляя их всем получателям. Кроме того, классы-прокси обеспечивают порядок и корректный источник пакетов, нумеруя исходящие пакеты и помечая их идентификатором матча. На приеме классы-прокси определяют номер очередного пакета и идентификатор матча. Если принятый пакет является самым новым и имеет корректный идентификатор, он будет принят, иначе пакет будет отброшен.

Существует вероятность, что на приеме класс-прокси получит мусорные данные. Это возможно в том случае, если контрольная сумма дейтаграммы не сможет обнаружить повреждение данных. В таком случае, попытка определить номер пакета может привести к тому, что номер следующего ожидаемого пакета будет много больше действительного. Это, в свою очередь, приведет к тому, что очередные пакеты от источника поврежденной дейтаграммы будут отбрасываться, вплоть до того момента, когда их порядковый номер не сравняется с номером некорректно принятой дейтаграммы. Чтобы отслеживать такого рода ошибки было принято решение оценивать разброс номеров пакетов. Для этого был реализован механизм, сохраняющий номера последних принятых пакетов и рассчитывающий их дисперсию. Дисперсия вычисляется по формуле:

$$D(x) = \frac{\sum_{i=1}^n (x_i - M)^2}{n},$$

где $D(x)$ – дисперсия случайной величины;

x_i – значения дискретной случайной величины;

M – среднее арифметическое дискретных случайных величин;

n – объем выборки.

В рассматриваемом случае в качестве случайной величины будут выступать номера пакетов. Если дисперсия номеров пакетов будет больше некоторого эталонного значения, можно сделать вывод, что были приняты некорректные данные. В таком случае в качестве номера очередного ожидаемого пакета будет выбрано минимальное значение из выборки.

Исходный код классов-прокси из библиотеки Network представлен в приложении Г.

Для сериализации и десериализации состояния игрового мира была разработана библиотека ECS.Serialization. Библиотека реализует контекст, в котором можно зарегистрировать сериализуемые компоненты. При регистрации компонентам назначается уникальный идентификатор. Для сериализуемых компонентов можно определить конвертеры, определяющие логику сериализации и десериализации. Конвертеры связываются с

компонентами с помощью специального атрибута. Компоненты, не зарегистрированные в контексте, игнорируются при сериализации.

Для того, чтобы применять полученные по сети данные к игровому миру и собирать актуальные данные перед отправкой были реализованы специальные классы для хоста (Snapshoter) и для клиента (Reconcilator). Исходный код данных классов представлен в приложении Д.

На рисунке 38 представлена диаграмма развертывания, демонстрирующая распределение компонентов приложения по физическим узлам.

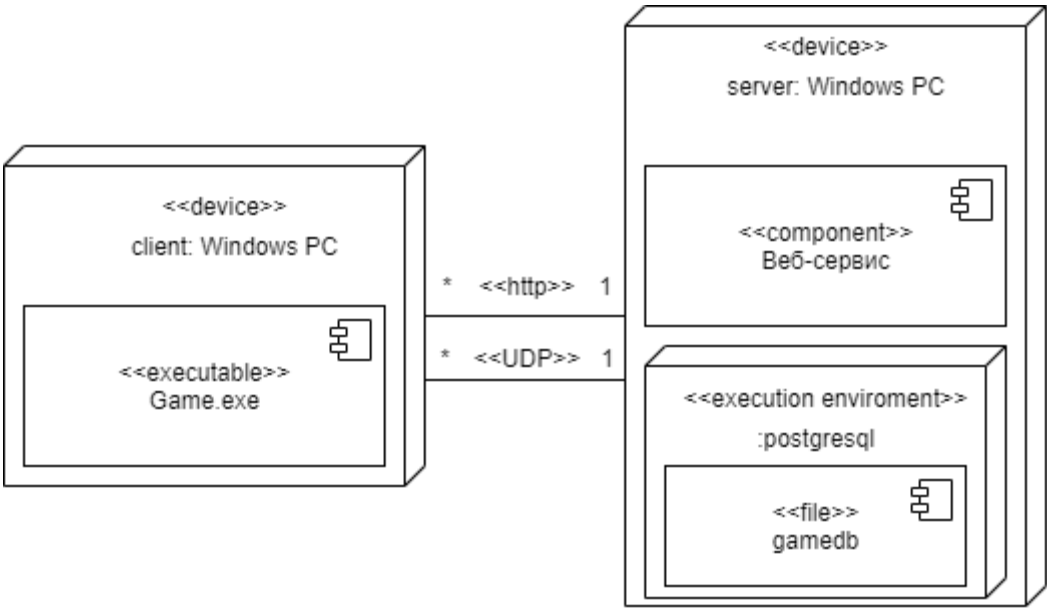


Рисунок 38 – Диаграмма развертывания

Для разработки клиента и сервера использовались персональные компьютеры под управлением ОС Windows. Однако благодаря тому, что игровой фреймворк Unity и фреймворк ASP.NET Core являются кроссплатформенными, в дальнейшем возможен переход на другие ОС.

4 Контроль качества программного обеспечения

4.1 Функциональное тестирование

Функциональное тестирование – тестирование, реализуемое в целях проверки способности ПО удовлетворять функциональным требованиям. Произведем функциональное тестирование веб-API и клиентского приложения.

4.1.1 Функциональное тестирование веб-API

Для тестирования веб-API был использован инструмент Postman. План функционального тестирования веб-API представлен в таблице 9.

Таблица 9 – План функционального тестирования веб-API

№	Описание теста	Запрос		Результат
		Метод	URI	
1	Зарегистрировать нового пользователя с корректным, не занятым логином и корректным паролем.	POST	api/account/registration	Тест пройден. Новый пользователь зарегистрирован. Код ответа HTTP – 200 (рисунок E.1).
2	Попытаться зарегистрировать нового пользователя с некорректным логином.			Тест пройден. Получено сообщение о неверном формате логина. Код ответа HTTP – 400 (рисунок E.2).
3	Попытаться зарегистрировать нового пользователя с некорректным паролем.			Тест пройден. Получено сообщение о некорректном пароле. Код ответа HTTP – 400 (рисунок E.3)
4	Попытаться зарегистрировать пользователя с занятым логином.			Тест пройден. Получено сообщение о том, что логин занят. Код ответа HTTP – 409 (рисунок E.4)
5	Войти в систему с использованием логина и пароля существующего пользователя.	POST	api/account/authorization	Тест пройден. Получен JWT токен и идентификатор пользователя. Код ответа HTTP – 200 (рисунок E.5)
6	Попытаться войти в систему под несуществующим логином.			Тест пройден. Код ответа HTTP – 401 (рисунок E.6)
7	Попытаться войти в систему с неверным паролем.			Тест пройден. Код ответа HTTP – 401 (рисунок E.7)

Продолжение таблицы 9

№	Описание теста	Запрос		Результат
		Метод	URI	
8	Проверить, вошел ли пользователь в систему с активной сессией.	GET	api/account/authorization	Тест пройден. Код ответа HTTP – 200 (рисунок Е.8)
9	Проверить, вошел ли пользователь с неактивной сессией.			Тест пройден. Код ответа HTTP – 401 (рисунок Е.9)

Тестирование непосредственно матчмейкера средствами Postman затруднительно, так как для этого требуется несколько одновременно активных пользовательских сессий. Кроме того, существуют жесткие ограничения на время совершения запросов матчмейкеру. Эти тестовые ситуации были проверены при тестировании клиентского приложения.

4.1.2 Функциональное тестирование клиентского приложения

План функционального тестирования клиентского приложения представлен в таблице 10.

Таблица 10 – План функционального тестирования клиентского приложения

№	Вариант использования	Тест	Результат
1	Зарегистрироваться в системе	Запустить сервер. Произвести регистрацию пользователя с корректным, не занятым логином и корректным паролем.	Тест пройден. Система оповестила об успешной регистрации (рисунок Ж.1).
2		Запустить сервер. Попытаться произвести регистрацию пользователя с некорректным логином.	Тест пройден. Система оповестила о некорректных символах в логине (рисунок Ж.2).
3		Запустить сервер. Попытаться произвести регистрацию пользователя с некорректным паролем.	Тест пройден. Система оповестила о некорректных символах в пароле (рисунок Ж.3).
4		Запустить сервер. Попытаться произвести регистрацию пользователя с занятым логином.	Тест пройден. Система оповестила о том, что логин занят (рисунок Ж.4).
5		Попытаться произвести регистрацию пользователя без логина.	Тест пройден. Система оповестила о том, что не указан логин (рисунок Ж.5).
6		Попытаться произвести регистрацию пользователя без пароля	Тест пройден. Система оповестила о том, что не указан пароль (рисунок Ж.6).
7		Не запускать сервер. Попытаться зарегистрировать пользователя с корректным логином и паролем.	Тест пройден. Система оповестила о том, что не удалось установить соединение с сервером (рисунок Ж.7).

Продолжение таблицы 10

№	Вариант использования	Тест	Результат
8	Войти в систему	Запустить сервер. Произвести вход в систему с использованием логина и пароля зарегистрированного пользователя.	Тест пройден. Система оповестила об успешном входе в систему (рисунок Ж.8).
9		Запустить сервер. Попытаться войти в систему под неверным логином.	Тест пройден. Система оповестила о том, что использован неверный логин или пароль (рисунок Ж.9).
10		Запустить сервер. Попытаться войти в систему под неверным паролем.	Тест пройден. Система оповестила о том, что использован неверный логин или пароль (рисунок Ж.10).
11		Не запускать сервер. Попытаться войти в систему.	Тест пройден. Система оповестила о том, что не удалось установить соединение с сервером (рисунок Ж.11).
12	Найти матч	Запустить сервер. Войти в систему за двух пользователей. Инициировать пользователями поиск матча.	Тест пройден. Оба пользователя попали в один матч (рисунок Ж.12).
13		Запустить сервер. Войти в систему. Инициировать поиск матча. Прервать поиск матча.	Тест пройден. Произошел переход в главное меню (рисунок Ж.13).
14		Запустить сервер. Войти в систему за двух пользователей. Инициировать пользователями поиск матча. Дождаться этапа соединения. Отменить поиск матча одним из пользователей.	Тест пройден. Система оповестила о разрыве соединения (рисунок Ж.14).
15		Запустить сервер. Дважды войти в систему за одного пользователя. Инициировать поиск матча.	Тест пройден. Система оповестила о том, что пользователь уже находится в очереди (рисунок Ж.15).
16		Не запускать сервер. Инициировать поиск матча.	Тест пройден. Система оповестила о том, что не удалось установить соединение с сервером (рисунок Ж.16).

Функциональное тестирование приложения показало корректность его работы.

4.2 Модульное тестирование

Модульное тестирование – вид тестирования ПО, проводящийся с целью проверить корректность работы отдельных модулей исходного кода программы. В качестве фреймворка для модульного тестирования был выбран фреймворк xUnit, так как его чаще всего используют в связке с ASP.NET Core.

Фреймворк xUnit спроектирован таким образом, чтобы реализуемые с его использованием тесты были максимально независимыми. Поэтому при

каждом новом запуске тестового метода фреймворк xUnit заново создает тестовый класс. Это не позволяет использовать поля и свойства тестового класса для разделения контекста между несколькими тестовыми методами. Однако для таких ситуаций фреймворк xUnit позволяет реализовать класс-контекст, который при каждом новом инстанцировании тестового класса будет передаваться в конструктор [28]. Для тестовых классов, в которых удобно иметь разделенный контекст, были реализованы соответствующие классы-контексты.

При написании модульных тестов использовался шаблон AAA (Arrange-Act-Assert), подразумевающий разделение тестовых методов на три секции. Первая секция (Arrange) нужна для инициализации конкретного теста. Следующая секция (Act) содержит вызов тестируемого метода. Последняя секция (Assert) содержит проверку полученных результатов [29]. В таблице 11 приведено описание некоторых модульных тестов. На рисунке 39 представлены результаты модельного тестирования.

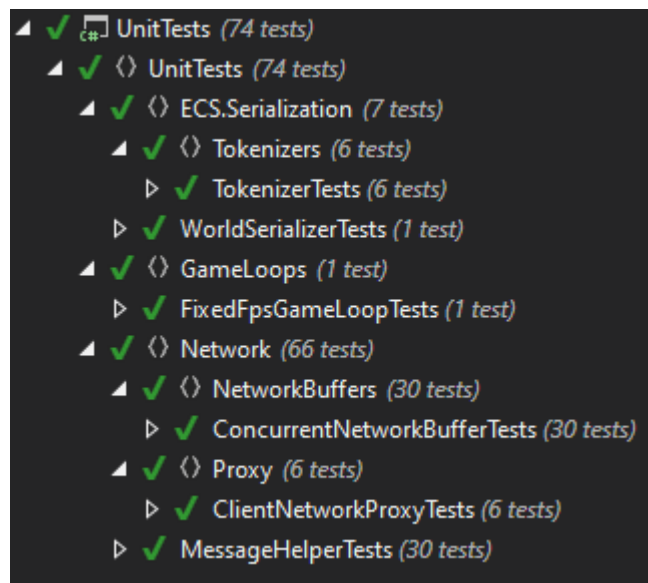


Рисунок 39 – Результаты модульного тестирования

С помощью модульных тестов были проверены наиболее нетривиальные в своей реализации методы и свойства.

Таблица 11 – Описание модульных тестов

Тестовый класс	Проверяемый класс	Тестовый метод	Проверяемый метод/свойство	Описание
MessageHelperTests	MessageHelper	GetMessageType_EmptyMessagesOfConcreteTypes_ReturnMessagesTypes	GetMessageType	Проверка корректности определения типов сообщений.
		ToByteArray_MessagesWithByteData_ReturnByteData	ToByteArray	Проверка корректности извлечения данных из сообщения в виде последовательности байт.
		ToString_MessageWithStringData_ReturnStringData	ToString	Проверка корректности извлечения данных из сообщения в виде строки.
ConcurrentNetworkBuffer Tests	ConcurrentNetwork Buffer	Read_MultipleMessagesWrite_ReturnWrittenMessages	Read	Проверка корректности чтения из буфера.
		ReadLast_MultipleMessagesWrite_ReturnLastWrittenMessage	ReadLast	Проверка корректности чтения последнего сообщения в буфере
		IsFull_MultipleMessagesWrite_ReturnCorrectFullStatus	IsFull	Проверка корректности определения состояния заполненности буфера.
		IsEmpty_MultipleMessagesWrite_ReturnCorrectEmptyStatus	IsEmpty	Проверка корректности определения пустоты буфера.
		Count_MultipleMessagesWrite_ReturnCorrectMessagesCount	Count	Проверка корректности определения количества сообщений в буфере
TokenizerTests	Tokenizer	Parse_ValidInputSequence_CorrectTokenSequence	Parse	Проверка корректности разбора корректных последовательностей.
		Parse_InvalidInputSequences_ThrowTokenizerException		Проверка реакции на некорректные последовательности.
WorldSerializerTests	WorldSerializer	Serialize_Entities_DeserializeSerializedEntity	Deserialize	Проверка корректности десериализации набора сущностей.

Модульное тестирование показало корректность работы проверяемых свойств иметодов.

4.3 Интеграционное тестирование

В отличие от модульных тестов интеграционные тесты не фокусируются на тестировании отдельных изолированных методов или свойств. Цель интеграционных тестов – проверить взаимодействие нескольких компонентов системы. Для того, чтобы проверить корректность классов, реализующих подключение игроков друг к другу, были реализованы интеграционные тесты.

По мимо разделяемого контекста в интеграционных тестах потребовалось обеспечить строгий порядок вызова тестовых методов. По умолчанию фреймворк xUnit вызывает методы в произвольной последовательности, мотивируя разработчиков писать максимально независимые тесты. Однако xUnit реализует механизмы, позволяющие упорядочивать вызовы тестовых методов, предоставляя возможность создавать собственные классы, определяющие этот самый порядок [30]. Для интеграционных тестов был реализован класс, упорядочивающий вызов тестовых методов, по номеру из специального атрибута.

В таблице 12 представлено описание интеграционных тестов.

Таблица 12 – Описание интеграционных тестов

Тестовый класс	Описание
MatchmakerServiceTests	Проверка работы сервиса-матчмейкера. Проверка работы матча, созданного матчмейкером.
MatchConnectorTests	Метод проверяет корректность взаимодействия коннектора для матча и матча.
HolePuncherTests	Метод проверяет корректность совместной работы коннектора для матчей, матча и коннектора, реализующего алгоритм UDP HolePunching.

На рисунке 40 представлены результаты интеграционного тестирования.

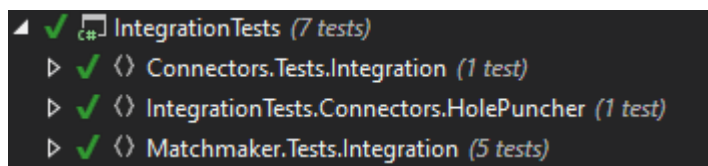


Рисунок 40 – Результаты интеграционного тестирования

Интеграционное тестирование позволило проверить корректность работы классов-коннекторов до их внедрения проект Unity.

4.4 Исследовательское тестирование

Из-за того, что игра имеет бесчисленное множество возможных состояний, четко определить сценарий, который бы смог охватить игровой процесс в полной мере, практически невозможно. Поэтому для тестирования игрового процесса было принято решение применить исследовательское тестирование. Исследовательское тестирование напоминает ad hoc тестирование, однако между этими подходами есть существенное отличие. В то время как ad hoc тестирование представляет собой поиск ошибок экспромтом, исследовательское тестирование предполагает вдумчивый подход к ad hoc тестированию, который с одной стороны дает возможность целенаправленно выявлять определенного рода ошибки, с другой, позволяет не составлять доскональную спецификацию всей системы. Таким образом, исследовательское тестирование определяет общую стратегию тестирования. То, какие тесты будут проведены в рамках этой общей стратегии, определяет сам тестирующий [31].

Для игрового процесса были определены глобальные тестовые случаи, покрывающие ту или иную игровую механику. В рамках этих тестовых случаев были выделены более конкретные тесты, покрывающие ключевые игровые ситуации. Результаты исследовательского тестирования представлены в таблице 13.

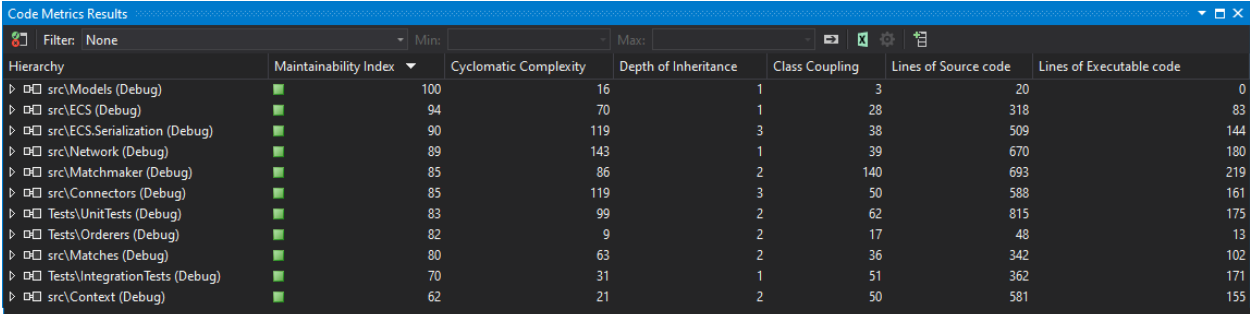
Таблица 13 – Описание исследовательских тестов

Игровая механика	Тест	Результат
Перемещение	Свободное перемещение во все доступные стороны.	Тест пройден. Персонаж корректно переместился (рисунки И.1 – И.2).
	Попытка перемещения за пределы карты.	Тест пройден. Персонаж не прошел за пределы карты.
	Попытка перемещения сквозь персонажа.	Тест пройден. Персонаж не смог пройти сквозь другого персонажа.
Очки здоровья/урон	Нанести урон другому персонажу	Тест пройден. Другой персонаж получил урон (рисунок И.3 – И.4).
	Получить урон	Тест пройден. Персонаж получил урон.
Боевая система	Наносить удары с разным размахом.	Тест пройден. Оружие корректно перемещается.
Конец игры	Победить.	Тест пройден. Игра оповестила о победе (рисунок И.5).
	Проиграть.	Тест пройден. Игра оповестила о поражении (рисунок И.6).

Исследовательское тестирование показало корректную работу клиентского приложения во время игры.

4.5 Метрики кода

Произведем расчет метрик кода для решения, в котором реализована серверная часть приложения, средствами Visual Studio (рисунок 41).



Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code	Lines of Executable code
src\Models (Debug)	100	16	1	3	20	0
src\ECS (Debug)	94	70	1	28	318	83
src\ECS.Serialization (Debug)	90	119	3	38	509	144
src\Network (Debug)	89	143	1	39	670	180
src\Matchmaker (Debug)	85	86	2	140	693	219
src\Connectors (Debug)	85	119	3	50	588	161
Tests\UnitTests (Debug)	83	99	2	62	815	175
Tests\Orderers (Debug)	82	9	2	17	48	13
src\Matches (Debug)	80	63	2	36	342	102
Tests\IntegrationTests (Debug)	70	31	1	51	362	171
src\Context (Debug)	62	21	2	50	581	155

Рисунок 41 – Результаты расчета метрик средствами Visual Studio

Средний индекс удобства поддержки (Maintainability Index) для решения составил 84, что говорит об относительной простоте поддержки кода.

Сложность организации циклов (Cyclomatic Complexity) позволяет определить количество различных путей в коде программы. Чем больше путей, тем больше требуется тестов для покрытия кода. Так как на рисунке 41 представлены метрики на уровне целых проектов, полученные результаты приемлемы.

Глубина наследования (Depth of Inheritance) позволяет определить, насколько далеко отстоит самый глубокий класс в цепочки наследования от базового класса. В код с малой глубиной наследования проще вносить изменения. Для проектов в решении глубина наследования не превышает 3, что является хорошим показателем.

Взаимосвязь классов (Class Coupling) позволяет оценить зависимость классов друг от друга. В сильно связанные компоненты труднее вносить изменения и их сложно повторно использовать. Для уровня проектов представленные значения взаимосвязи являются приемлемыми [32].

Исходя из метрик, полученных средствами Visual Studio, можно сделать вывод, что разработанный код имеет достаточно высокий уровень качества.

Произведем оценку качества одного из ключевых методов программы. Метод представлен в листинге 2.

Листинг 2 – Метод ConnectionFrame

```
private async Task ConnectionFrame()
{
    await SendPackets();

    while (_udpClient.Available > 0)
    {
        try
        {
            var result = await _udpClient.ReceiveAsync();
            ProcessPacket(result.Buffer, result.RemoteEndPoint);
        }
        catch (SocketException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

Данный метод находится в классе HolePuncher, реализующем алгоритм UDP Hole Punching. В данном методе определена логика соединения клиентов. Произведем для данного метода расчет цикломатической метрики Мак-Кейба и метрик Холстеда.

4.5.1 Цикломатическая метрика Мак-Кейба

Цикломатическая метрика Мак-Кейба (цикломатическая сложность) позволяет оценить структурную сложность программы. Для того, чтобы оценить цикломатическую сложность метода ConnectionFrame построим граф потока управления данного метода (рисунок 42).

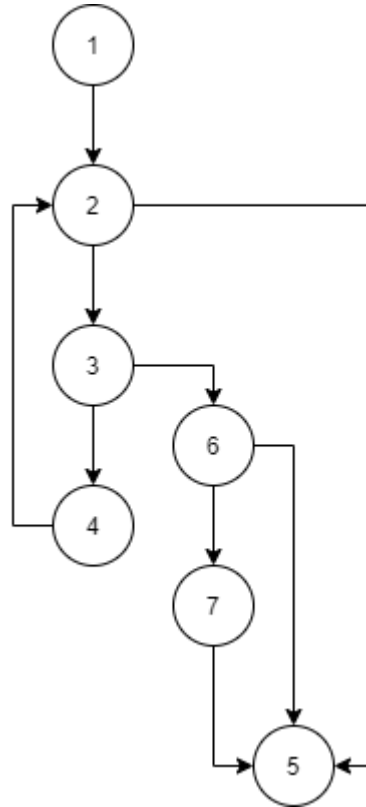


Рисунок 42 – Граф потока управления метода ConnectionFrame

Цикломатическая сложность вычисляется по следующей формуле:

$$V(G) = e - n + 2p,$$

где e – количество дуг;

n – количество вершин;

p – число компонент связности.

Граф потока управления метода ConnectionFrame содержит 7 вершин и 9 дуг. Подставим полученные значения: $V(G) = 9 - 7 + 2 = 4$.

Полученное значение цикломатической сложности меньше 10, что говорит о структурной простоте метода ConnectionFrame.

4.5.2 Метрика Холстеда

Для расчета метрик Холстеда необходимо измерить следующие характеристики программы:

n_1 – число уникальных операторов программы, включая символы-разделители, имена процедур и знаки операций (словарь операторов);

n_2 – число уникальных операндов программы, включая константы и имена переменных (словарь операндов);

N_1 – общее число операторов в программе;

N_2 – общее число операндов в программе.

Характеристики метода ConnectionFrame представлены в таблице 14.

Таблица 14 – Характеристики метода ConnectionFrame

Операторы			Операнды		
№	Оператор	Количество	№	Операнд	Количество
1	await	2	1	_udpClient	1
2	SendPackets	1	2	_udpClient.Available	1
3	()	6	3	result	1
4	;	4	4	result.Buffer	1
5	while	1	5	result.RemoteEndPoint	1
6	>	1	6	e	1
7	{}	3	7	e.Message	1
8	try	1			
9	var	1			
10	=	1			
11	ReceiveAsync	1			
12	ProcessPacket	1			
13	,	1			
14	catch	1			
15	WriteLine	1			
Всего:	$n_1 = 15$	$N_1 = 26$	Всего:	$n_2 = 7$	$N_2 = 7$

Словарь программы рассчитывается по формуле:

$$n = n_1 + n_2 = 15 + 7 = 22.$$

Длина программы рассчитывается по формуле:

$$N = N_1 + N_2 = 26 + 7 = 33.$$

Объем программы рассчитывается по формуле:

$$HV = N \cdot \log_2(n) = 33 \cdot 4.46 = 147.18.$$

Сложность программы рассчитывается по формуле:

$$HD = \left(\frac{n1}{2}\right) \cdot \left(\frac{N2}{n2}\right) = \left(\frac{15}{2}\right) \cdot \left(\frac{7}{7}\right) = 7.5.$$

Полученные метрики говорят о том, что код метода `ConnectionFrame` лаконичен и прост.

Заключение

В рамках выпускной квалификационной работы было разработано игровое приложение с многопользовательским режимом игры. Для игрового приложения был реализован сетевой код, позволяющий игроку участвовать в игровых сессиях со случайными напарниками.

Был произведен анализ предметной области, в ходе которого было произведено сравнение игровых приложений, аналогичных разрабатываемому. Для рассмотренных аналогов были выделены их ключевые особенности и существующие недостатки. В результате анализа аналогов было принято решение разработать собственное игровое приложение с многопользовательским режимом игры. Для разрабатываемого игрового приложения в целом и сетевого кода в частности были определены функциональные и нефункциональные требования.

Было произведено планирование разработки и оценка бюджета проекта. Длительность реализации проекта составила 96 дней, а предполагаемые затраты – 236 880 руб., что говорит об экономической целесообразности разработки.

На этапе проектирования программных средств был применен язык моделирования UML. В качестве среды для построения UML диаграмм была использована программа «Draw.io»

Для реализации программных средств использовался язык программирования C#. В качестве среды разработки была использована IDE Visual Studio. Для реализации игрового приложения был использован фреймворк Unity.

Было проведено функциональное, модульное, интеграционное и исследовательское тестирование разработанного приложения. На основе результатов тестирования можно сделать вывод, что разработанное приложение соответствует поставленным требованиям и работает корректно.

Таким образом, в результате выполнения выпускной квалификационной работы были разработаны программные средства, отвечающие всем требованиям технического задания.

Список использованных источников

1. Ильянов Д.С., Источник роста экономики в XXI веке: индустрия видеоигр / Ильянов Д.С., Чернышева Т.К., Юревич М.А. // Теоретическая и прикладная экономика. – 2020. – №3. DOI: 10.25136/2409-8647.2020.3.31693 URL:https://nbpublish.com/library_read_article.php?id=31693 (дата обращения 15.04.2021)
2. Глейзер, Дж. Многопользовательские игры. Разработка сетевых приложений.: пер. с англ. / Дж. Гейзер, С. Мадхав. – СПб.: Питер, 2017. – 368 с.
3. PLATO computer-based education system // Encyclopedia Britannica | Britannica [Электронный ресурс] – URL: <https://www.britannica.com/topic/PLATO-education-system> (дата обращения: 18.04.2021) (дата обращения 15.04.2021)
4. Most Popular PC Games | Global // Newzoo [Электронный ресурс] – URL: <https://newzoo.com/insights/rankings/top-20-pc-games/> (дата обращения 16.04.2021)
5. The World's 2.7 Billion Gamers Will Spend \$159.3 Billion on Games in 2020; The Market Will Surpass \$200 Billion by 2023 // Newzoo [Электронный ресурс] – URL: <https://newzoo.com/insights/articles/newzoo-games-market-numbers-revenues-and-audience-2020-2023/> (дата обращения 16.04.2021)
6. Gaming: The Most Lucrative Entertainment Industry By Far // Statista [Электронный ресурс] – URL: <https://www.statista.com/chart/22392/global-revenue-of-selected-entertainment-industry-sectors/> (дата обращения 17.04.2021)
7. Registered users of Fortnite worldwide from August 2017 to May 2020 // Statista [Электронный ресурс] – URL: <https://www.statista.com/statistics/746230/fortnite-players/> (дата обращения 17.04.2021)

8. Monthly number of peak concurrent players of Grand Theft Auto V on Steam worldwide from January 2016 to March 2021 // Statista [Электронный ресурс] – URL: <https://www.statista.com/statistics/980448/gtav-number-players-steam/#:~:text=The%20statistic%20shows%20information%20on,thousand%20concurrent%20players%20on%20Steam> (дата обращения 17.04.2021)
9. Counter-Strike: Global Offensive // Steam Charts [Электронный ресурс] – URL: <https://steamcharts.com/app/730> (дата обращения 17.04.2021)
10. Portal 2 // Steam Charts [Электронный ресурс] – URL: <https://steamcharts.com/app/620> (дата обращения 17.04.2021)
11. Borderlands 3 // Steam Charts [Электронный ресурс] – URL: <https://steamcharts.com/app/397540> (дата обращения 17.04.2021)
12. It Takes Two // Steam Charts [Электронный ресурс] – URL: <https://steamcharts.com/app/1426210> (дата обращения 17.04.2021)
13. Salt and Sanctuary // Steam [Электронный ресурс] – URL: https://store.steampowered.com/app/283640/Salt_and_Sanctuary/?l=russian (дата обращения 18.04.2021)
14. Risk of Rain // Steam [Электронный ресурс] – URL: https://store.steampowered.com/app/248820/Risk_of_Rain/ (дата обращения 18.04.2021)
15. Furi // Steam [Электронный ресурс] – URL: <https://store.steampowered.com/app/423230/Furi/> (дата обращения 18.04.2021)
16. Most important gaming platforms for future growth according to game developers worldwide as of March 2019 // Statista [Электронный ресурс] – URL: <https://www.statista.com/statistics/1128100/platforms-game-developers-future-growth/> (дата обращения 19.04.2021)
17. Share of Steam (gaming platform) users in October 2020, by operating system used // Statista [Электронный ресурс] – URL: <https://www.statista.com/statistics/265033/proportion-of-operating-systems-used-on-the-online-gaming-platform-steam/> (дата обращения 19.04.2021)

18. UDP vs. TCP // Gaffer On Games [Электронный ресурс] – URL: https://gafferongames.com/post/udp_vs_tcp/ (дата обращения 23.04.2021)
19. Учебник. Создание веб-API с помощью ASP.NET Core. // MSDN [Электронный ресурс] <https://docs.microsoft.com/ru-ru/aspnet/core/tutorials/first-web-api?view=aspnetcore-5.0&tabs=visual-studio> (дата обращения 25.04.2021)
20. Внедрение зависимостей в ASP.NET Core // MSDN [Электронный ресурс] – URL: <https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0> (дата обращения 25.04.2021)
21. What is an Entity Component System architecture for game development? // Richard Lord [Электронный ресурс] – URL: <https://www.richardlord.net/> (дата обращения 26.04.2021)
22. Unity, ECS и все-все-все // Хабр [Электронный ресурс] – URL: <https://habr.com/ru/post/358108/> (дата обращения 26.04.2021)
23. richardlord/Ash // GitHub [Электронный ресурс] – URL: <https://github.com/richardlord/Ash> (дата обращения 27.04.2021)
24. DavidArno/Ash.NET // GitHub [Электронный ресурс] – URL: <https://github.com/DavidArno/Ash.NET> (дата обращения 27.04.2021)
25. Ford B., Pear-to-Pear Communication Across Network Address Translation / Ford B., Srisuresh P. // 2005 USENIX Annual Technical Conference. URL: https://www.usenix.org/legacy/event/usenix05/tech/general/full_papers/ford/ford.pdf (дата обращения 28.04.2021)
26. Code First // Entity Framework Core [Электронный ресурс] – URL: <https://entityframeworkcore.com/approach-code-first> (дата обращения 08.05.2021)
27. Рабочий процесс Gitflow Workflow // Atlassian [Электронный ресурс] – URL: <https://www.atlassian.com/ru/git/tutorials/comparing-workflows/gitflow-workflow> (дата обращения 08.05.2021)

28. Unit.net » Documentation // Shared Context between Tests [Электронный ресурс] – URL: <https://xunit.net/docs/shared-context> (дата обращения 27.05.2021)
29. Paulo Gomes // Unit Testing and the Arrange, Act and Assert (AAA) Pattern [Электронный ресурс] – URL: <https://medium.com/@pjbfgf/title-testing-code-ocd-and-the-aaa-pattern-df453975ab80> (дата обращения 27.05.2021)
30. Порядок модульных тестов // MSDN [Электронный ресурс] – URL: <https://docs.microsoft.com/ru-ru/dotnet/core/testing/order-unit-tests?pivots=xunit> (дата обращения 28.05.2021)
31. Exploratory Testing // Satisfice [Электронный ресурс] – URL: <https://www.satisfice.com/exploratory-testing> (дата обращения 29.05.2021)
32. Code metrix values // MSDN [Электронный ресурс] – URL: <https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2019> (дата обращения 30.05.2021)

Приложение А
ГЛОССАРИЙ
(справочное)

Аркада – жанр компьютерных игр, характеризующийся коротким по времени, но интенсивным игровым процессом. В узком смысле аркадными называются игры для аркадных игровых автоматов.

Приключенческая игра – жанр компьютерных игр, в котором основной упор делается на сюжет. Для жанра характерны продуманный сюжет, большое количество диалогов, персонажи с детально прописанными характерами. С точки зрения игрового процесса в играх данного жанра большое внимание уделяется разного рода загадкам.

Текстовая приключенческая игра – разновидность приключенческих игр, в которых взаимодействие игрока с игрой происходит посредством текста.

Экшен – жанр компьютерных игр, бросающих вызов реакции и физическим возможностям игрока.

RPG – жанр компьютерных игр, основанный на элементах игрового процесса традиционных настольных ролевых игр.

Action/RPG – Ролевая игра в жанре «экшен».

Открытый мир – термин в компьютерных играх, обозначающий виртуальные миры, которые игрок может свободно исследовать.

Симулятор – жанр компьютерных игр, стремящихся воспроизвести в формате видеоигры какую-либо деятельность или явление из реальной жизни.

Спортивный симулятор – симулятор спортивных состязаний.

Файтинг – жанр компьютерных игр, имитирующих рукопашный бой малого числа персонажей в пределах ограниченного пространства, называемого арена.

Игровой мир – совокупность всех игровых объектов и их свойств в рамках игровой сессии.

Сетевой код – обширный термин, применяемый по отношению сетевой части многопользовательских игр. В общем случае под понятием «сетевой код» понимают все аспекты игрового приложения, так или иначе связанные с многопользовательским режимом игры. Объединение нескольких игроков в рамках одной игровой сессии, организация обмена данными между этими

игроками, оптимизация сетевого трафика, компенсация задержек, механизмы предсказаний, репликация объектов, выбор сетевой архитектуры – все это задачи, решаемые сетевым кодом.

Free-to-Play – модель распространения видеоигр, позволяющая пользователям играть в игру без внесения денежных средств.

Экшен – жанр компьютерных игр, бросающих вызов реакции и физическим возможностям игрока.

Игровое дополнение – дополнительный материал к существующей самостоятельной видеоигре.

Геймплей, или игровой процесс – способ игрока взаимодействия с игровым миром. Понятие «геймплей» включает себя с одной стороны набор доступных для игрока действий, с другой, реакцию игрового мира на действия, совершенные игроком.

Комьюнити – сообщество людей с общими интересами.

Игровая сессия – отрезок времени, в пределах которого происходит игровой процесс.

Предсказания – один из возможных способов компенсации сетевой задержки, подразумевающий, что клиенты просчитывают последующие состояния игрового мира, не дожидаясь информации об актуальном состоянии. Когда со стороны сервера приходит сообщение, содержащее состояние мира, клиент сверяет его с состоянием своего мира и при необходимости корректирует его.

Репликация объектов – акт передачи состояния объектов игрового мира.

Геймдизайн – процесс создания формы и содержания геймплея разрабатываемой игры.

Босс – игровой персонаж-противник, которого значительно сложнее победить, чем обычных врагов.

Игровая механика – набор правил и способов, реализующий определённым образом некоторую часть интерактивного взаимодействия игрока и игры.

Контрольная точка – игровая механика, позволяющая сохранять игровой прогресс в определенных местах игры.

Боевая система – набор игровых механик, определяющих способы ведения боя.

Платформер – жанр видеоигр, основу геймплея которых составляет перемещение в качестве игрового персонажа по уровню. Платформеры отличаются тем, что для продвижения по игровым уровням игроку часто приходится использовать прыжки в качестве основного инструмента перемещения.

Roguelike – жанр видеоигр, основными особенностями которых является процедурно генерируемые уровни, пошаговость и необратимость смерти персонажа.

Шутер – жанр видеоигр, основу геймплея которых составляют сражения с противниками при помощи огнестрельного, или любого другого оружия, способного поразить противника на расстоянии.

Shoot'em up – жанр видеоигр, представляющий собой одну из разновидностей шутера. Не существует единого мнения относительно геймплейных элементов, характеризующий данный жанр, однако в наиболее общем случае к играм жанра Shoot'em up можно отнести игры, в которых герою необходимо противостоять большому количеству врагов, стреляя по ним, и уворачиваться от вражеского огня.

Hack and slash – жанр видеоигр, основу геймплея которых составляют сражения с противниками с помощью оружия в ближнем бою.

Bullet hell – поджанр игр жанра шутер, особенностью которых является чрезвычайно большое количество вражеских снарядов.

Искусственный интеллект противников – набор программных методик, которые используются в компьютерных играх для создания иллюзии интеллекта в поведении персонажей, управляемых компьютером.

Матчмейкер – механизм, ответственный за создание игровых сессий в многопользовательских играх.

Время приёма-передачи (Round Trip Time, RTT) - это время, которое требуется для отправки пакета данных в пункт назначения, плюс время, которое требуется для подтверждения того, что этот пакет был получен обратно.

Внедрение зависимостей – процесс предоставления внешней зависимости программному компоненту.

Конечная точка – комбинация ip адреса и порта.

NAT – это механизм в сетях TCP/IP, позволяющий преобразовывать IP-адреса транзитных пакетов.

Приложение Б
Исходный код веб-API
(рекомендуемое)

Листинг Б.1 – MatchmakerController.cs

```

using System.Net;
using System.Net.Sockets;
using Matchmaker.Helpers;
using Matchmaker.Services;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using Network;
using Network.Messages;

namespace Matchmaker.Controllers
{
    [Route("api/[controller]")]
    public class MatchmakerController : ControllerBase
    {
        private static readonly string LocalIp =
NetworkHelper.GetLocalIPAddress();

        private readonly IMatchmakerService _matchmakerService;
        private readonly ILogger<MatchmakerController> _logger;

        public MatchmakerController(
            IMatchmakerService matchmakerService,
            ILogger<MatchmakerController> logger)
        {
            _matchmakerService = matchmakerService;
            _logger = logger;
        }

        [Authorize]
        [HttpPut("queue")]
        public bool Enqueue([FromBody]ClientEndPoint privateEndPoint)
        {
            var userId = User.GetName();
            _logger.LogInformation($"Запрос на постановку в очередь (id
пользователя: {userId}).");
            var publicEndPoint = new ClientEndPoint(GetClientIp(), 0);
            var endPoints = new ClientEndPoints(publicEndPoint,
privateEndPoint);
            return _matchmakerService.Enqueue(userId, endPoints);
        }

        [Authorize]
        [HttpGet("status")]
        public UserStatus GetStatus()
        {
            var userId = User.GetName();
            _logger.LogInformation($"Запрос на получение статуса (id
пользователя: {userId}).");
            return _matchmakerService.GetStatus(userId);
        }

        [Authorize]
        [HttpGet("match")]
        public int? GetMatch()
        {
            var userId = User.GetName();
            _logger.LogInformation($"Запрос на получение матча (id
пользователя: {userId}).");

```

```

        return _matchmakerService.GetMatch(userId);
    }

    [Authorize]
    [HttpDelete("queue")]
    public bool Remove()
    {
        var userId = User.GetName();
        _logger.LogInformation($"Запрос на выход из очереди (id
пользователя: {userId}).");
        return _matchmakerService.Remove(userId);
    }

    private string GetClientIp()
    {
        var clientIp =
HttpContext.Connection.RemoteIpAddress.AddressFamily ==
AddressFamily.InterNetworkV6
        ? LocalIp // Если сервер запущен локально, то RemoteIpAddress
представлены в виде IPv6 адреса. Заменяем его на IPv4.
        : HttpContext.Connection.RemoteIpAddress.ToString();

        // Если запрос пришел с компьютера, на котором запущен сервер,
заменяем ip на 127.0.0.1.
        return clientIp == LocalIp ? IPAddress.Loopback.ToString() :
clientIp;
    }
}

```

Листинг Б.2 – MatchmakerService.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using Matches;
using Matchmaker.Factories;
using Microsoft.Extensions.Logging;
using Network;
using Network.Messages;

namespace Matchmaker.Services
{
    public class MatchmakerService : IMatchmakerService, IDisposable
    {
        private class ClientRecord
        {
            public ClientEndpoints EndPoints { get; }
            public long LastRequestTime { get; set; }

            public ClientRecord(ClientEndpoints endPoints, long
lastRequestTime)
            {
                EndPoints = endPoints;
                LastRequestTime = lastRequestTime;
            }
        }

        private const int MatchmakerDelay = 1000 / 60;
        private const int CleaningDelay = 5000;
    }
}

```

```

private const int AllowedDelta = CleaningDelay;

private readonly CancellationTokenSource _tokenSource;
private readonly Dictionary<string, ClientRecord> _playersRecords =
new Dictionary<string, ClientRecord>();
private readonly Dictionary<string, IMatch> _playerToMatch = new
Dictionary<string, IMatch>();

private readonly IMatchFactory _matchFactory;
private readonly ILogger<MatchmakerService> _logger;

private bool _disposed;

public int PlayersPerMatch => 2;

public MatchmakerService(
    IMatchFactory matchFactory,
    ILogger<MatchmakerService> logger)
{
    _matchFactory = matchFactory;
    _logger = logger;
    _tokenSource = new CancellationTokenSource();

    Task.Run(
        () => DelayedLoop(MatchmakingFrame, MatchmakerDelay,
            _tokenSource.Token),
        _tokenSource.Token);

    Task.Run(
        () => DelayedLoop(CleaningFrame, CleaningDelay,
            _tokenSource.Token),
        _tokenSource.Token);
}

public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

public bool Enqueue(string userId, ClientEndpoints endPoints)
{
    _logger.LogInformation($"Постановка в очередь (id пользователя:
{userId}).");

    lock (_playersRecords)
    {
        if (_playersRecords.ContainsKey(userId))
        {
            _logger.LogInformation($"Пользователь уже в очереди (id
пользователя: {userId}).");
            return false;
        }

        _playersRecords[userId] = new ClientRecord(endPoints,
            GetCurrentTime());
        _logger.LogInformation($"Пользователь добавлен в очередь (id
пользователя: {userId}).");
        return true;
    }
}

```

```

        public UserStatus GetStatus(string userId)
        {
            _logger.LogInformation($"Запрос статуса (id пользователя:
{userId}).");

            lock (_playersRecords)
            {
                if (_playersRecords.TryGetValue(userId, out var record))
                {
                    _logger.LogInformation($"Пользователь ожидает (id
пользователя: {userId}).");
                    record.LastRequestTime = GetCurrentTime();
                    return UserStatus.Wait;
                }
            }

            lock (_playerToMatch)
            {
                if (_playerToMatch.ContainsKey(userId))
                {
                    _logger.LogInformation($"Пользователь добавлен в матч (id
пользователя: {userId}).");
                    return UserStatus.Connected;
                }
            }

            _logger.LogInformation($"Пользователь отсутствует (id
пользователя: {userId}).");
            return UserStatus.Absent;
        }

        public int? GetMatch(string userId)
        {
            _logger.LogInformation($"Запрос матча (id пользователя:
{userId}).");

            lock (_playerToMatch)
            {
                if (_playerToMatch.TryGetValue(userId, out var match))
                {
                    _logger.LogInformation($"Матч найден (id пользователя:
{userId}).");
                    return match.Port;
                }

                _logger.LogInformation($"Матч не найден (id пользователя:
{userId}).");
                return null;
            }
        }

        public bool Remove(string userId)
        {
            _logger.LogInformation($"Выход из очереди (id пользователя:
{userId}).");

            lock (_playersRecords)
            {
                return _playersRecords.Remove(userId);
            }
        }

        protected virtual void Dispose(bool disposing)

```

```

    {
        if (_disposed) return;

        if (disposing)
        {
            _tokenSource.Cancel();
            _tokenSource.Dispose();
            _disposed = true;
        }
    }

    private async Task DelayedLoop(Action action, int delay,
Cancellation token cancellationToken)
    {
        while (true)
        {
            cancellationToken.ThrowIfCancellationRequested();
            action();
            await Task.Delay(delay, cancellationToken);
        }
    }

    private void MatchmakingFrame()
    {
        lock (_playersRecords)
        {
            while (_playersRecords.Count >= PlayersPerMatch)
            {
                _logger.LogInformation("Создаем матч.");
                var playersPairs =
_playersRecords.Take(PlayersPerMatch).ToList();
                var playersEndPoints = playersPairs.Select(o =>
o.Value.EndPoints).ToList();
                var match = StartNewMatch(playersEndPoints);

                lock (_playerToMatch)
                {
                    playersPairs.ForEach(o => _playerToMatch[o.Key] =
match);
                }

                playersPairs.ForEach(o => _playersRecords.Remove(o.Key));
            }
        }
    }

    private IMatch StartNewMatch(ICollection<ClientEndPoints>
playersEndPoints)
    {
        var match = _matchFactory.CreateMatch(playersEndPoints);
        match.MatchStarted += RemoveAssociatedPlayers;

        Task.Run(async () => await match.WorkAsync(_tokenSource.Token),
_tokenSource.Token)
        .ContinueWith(t =>
        {
            if (!t.IsCompletedSuccessfully)
            {
                RemoveAssociatedPlayers(match);
            }

            match.Dispose();
        });
    }

```

```

        _logger.LogInformation($"Матч запущен на порту {match.Port}.");
        return match;
    }

    private void RemoveAssociatedPlayers(IMatch match)
    {
        lock (_playerToMatch)
        {
            _playerToMatch
                .Where(pair => pair.Value == match)
                .ToList()
                .ForEach(pair => _playerToMatch.Remove(pair.Key));
        }
    }

    private void CleaningFrame()
    {
        lock (_playersRecords)
        {
            long currentTime = GetCurrentTime();
            _playersRecords
                .Where(record => currentTime -
record.Value.LastRequestTime > AllowedDelta)
                .ToList()
                .ForEach(pair => _playersRecords.Remove(pair.Key));
        }
    }

    private long GetCurrentTime()
        => ((DateTimeOffset)DateTime.Now).ToUnixTimeMilliseconds();
    }
}

```

Приложение В
Исходный код классов-коннекторов
(рекомендуемое)

Листинг В.1 – MatchmakerConnector.cs

```

using System.Net.Http;
using System.Net.Http.Headers;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Network.Messages;

namespace Connectors.MatchmakerConnectors
{
    public class MatchmakerConnector : IMatchmakerConnector
    {
        private const int LoopDelay = 1000;
        private const int TimeForNotification = 1000;

        private bool _isRun;
        private HttpClient _client;
        private CancellationToken _cancellationToken;

        public ClientEndPoint PrivateEndPoint { get; private set; }
        public string Host { get; private set; }

        internal MatchmakerConnectorStateBase State { get; set; }
        internal int? MatchPort { get; set; }

        public async Task<int> ConnectAsync(
            ClientEndPoint privateEndPoint,
            string host,
            string bearerToken,
            CancellationToken cancellationToken = default)
        {
            PrivateEndPoint = privateEndPoint;
            Host = host;
            _cancellationToken = cancellationToken;
            _client = new HttpClient();
            _client.DefaultRequestHeaders.Authorization
                = new AuthenticationHeaderValue("Bearer", bearerToken);
            State = new EnqueueState(this);
            await ConnectionLoopAsync();
            return MatchPort.Value;
        }

        internal async Task<HttpResponseMessage> PutAsync(string uri, string
json)
            => await _client.PutAsync(
                uri,
                new StringContent(json, Encoding.UTF8, "application/json"),
                _cancellationToken);

        internal async Task<HttpResponseMessage> GetAsync(string uri)
            => await _client.GetAsync(uri, _cancellationToken);

        internal void FinishConnection() => _isRun = false;

        private async Task ConnectionLoopAsync()
        {
            _isRun = true;

```

```

        try
        {
            while (!_isRun)
            {
                _cancellationToken.ThrowIfCancellationRequested();
                await State.ConnectAsync();
                await Task.Delay(LoopDelay, _cancellationToken);
            }
        }
        finally
        {
            if (_cancellationToken.IsCancellationRequested)
            {
                try
                {
                    await NotifyAboutCancellation(TimeForNotification);
                }
                catch (TaskCanceledException) { }
            }

            _client.Dispose();
        }
    }

    private async Task NotifyAboutCancellation(int timeout)
    {
        var tokenSource = new CancellationTokenSource();
        tokenSource.CancelAfter(timeout);
        await _client.DeleteAsync($"{Host}/api/queue",
tokenSource.Token);
    }
}

```

Листинг В.2 – MatchmakerConnectorStateBase.cs

```

using System.Threading.Tasks;

namespace Connectors.MatchmakerConnectors
{
    internal abstract class MatchmakerConnectorStateBase
    {
        protected MatchmakerConnector Context { get; }

        public MatchmakerConnectorStateBase(MatchmakerConnector context)
            => Context = context;

        public abstract Task ConnectAsync();
    }
}

```

Листинг В.3 – EnqueueState.cs

```

using System.Threading.Tasks;
using Newtonsoft.Json;

```

```

namespace Connectors.MatchmakerConnectors
{
    internal class EnqueueState : MatchmakerConnectorStateBase
    {
        private readonly string _json;

        public EnqueueState(MatchmakerConnector context)
            : base(context)
            => _json = JsonConvert.SerializeObject(Context.PrivateEndPoint);

        public override async Task ConnectAsync()
        {
            var response = await
Context.PutAsync($"{Context.Host}/api/matchmaker/queue", _json);

            if (response.IsSuccessStatusCode)
            {
                var data = await response.Content.ReadAsStringAsync();

                if (bool.Parse(data))
                {
                    Context.State = new WaitState(Context);
                }
                else
                {
                    throw new ConnectorException("Пользователь уже находится в
очереди.");
                }
            }
            else
            {
                throw new HttpConnectorException("Не удалось встать в
очередь", response.StatusCode);
            }
        }
    }
}

```

Листинг В.4 – WaitState.cs

```

using System.Threading.Tasks;
using Network;

namespace Connectors.MatchmakerConnectors
{
    internal class WaitState : MatchmakerConnectorStateBase
    {
        public WaitState(MatchmakerConnector context)
            : base(context)
        { }
    }
}

```

```

public override async Task ConnectAsync()
{
    switch (await GetUserStatus())
    {
        case UserStatus.Connected:
            Context.State = new GetMatchState(Context);
            break;
        case UserStatus.Absent:
            Context.State = new EnqueueState(Context);
            break;
    }
}

private async Task<UserStatus> GetUserStatus()
{
    var response = await
Context.GetAsync($"{Context.Host}/api/matchmaker/status");

    if (!response.IsSuccessStatusCode)
    {
        throw new HttpConnectorException("Не удалось получить статус
пользователя", response.StatusCode);
    }

    var data = await response.Content.ReadAsStringAsync();
    return (UserStatus)int.Parse(data);
}
}

```

Листинг В.5 – GetMatchState.cs

```

using System.Threading.Tasks;

namespace Connectors.MatchmakerConnectors
{
    internal class GetMatchState : MatchmakerConnectorStateBase
    {
        public GetMatchState(MatchmakerConnector context)
            : base(context)
        { }

        public override async Task ConnectAsync()
        {
            Context.MatchPort = await GetMatchPort();

            if (Context.MatchPort == null)
            {
                Context.State = new WaitState(Context);
            }
            else
            {
                Context.FinishConnection();
            }
        }
    }
}

```

```

        private async Task<int?> GetMatchPort()
        {
            var response = await
Context.GetAsync($"{Context.Host}/api/matchmaker/match");

            if (!response.IsSuccessStatusCode)
            {
                throw new HttpConnectorException("Не удалось получить порт
матча", response.StatusCode);
            }

            var data = await response.Content.ReadAsStringAsync();
            return int.TryParse(data, out var number) ? (int?)number : null;
        }
    }
}

```

Листинг В.6 – MatchConnector.cs

```

using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Threading.Tasks;
using Network;
using Network.Messages;

namespace Connectors.MatchConnectors
{
    public class MatchConnector : IMatchConnector
    {
        private const int LoopDelay = 100;
        private const int ReceiveTimeout = 100;

        private bool _isRun;
        private int _currentAttempts;
        private UdpClient _udpClient;
        private CancellationToken _cancellationToken;

        public int MaxAttempts { get; set; } = 10;
        public string ServerIp { get; private set; }
        public int ServerPort { get; private set; }
        internal string Ip { get; private set; }
        internal int Port { get; private set; }
        internal ConnectionMessage ConnectionMessage { get; set; }
        internal MatchConnectorStateBase State { get; set; }

        public async Task<ConnectionMessage> ConnectAsync(
            UdpClient client,
            string serverIp,
            int serverPort,
            CancellationToken cancellationToken = default)
        {
            ServerIp = serverIp;
            ServerPort = serverPort;
            _udpClient = client;
            _cancellationToken = cancellationToken;
            Ip = NetworkHelper.GetLocalIPAddress();
            Port = client.GetPort();
            State = new HelloMatchState(this);
            await ConnectionLoopAsync();
            return ConnectionMessage;
        }
    }
}

```

```

internal async Task SendMessageAsync(byte[] message)
    => await _udpClient.SendAsync(message, message.Length, ServerIp,
ServerPort);

internal void FinishConnection() => _isRun = false;

private async Task ConnectionLoopAsync()
{
    _isRun = true;

    while (_isRun)
    {
        _cancellationToken.ThrowIfCancellationRequested();
        await ConnectionFrameAsync();
        await Task.Delay(LoopDelay, _cancellationToken);
    }
}

private async Task ConnectionFrameAsync()
{
    await State.SendMessageAsync();

    while (_isRun && _udpClient.Available > 0)
    {
        if (TryReceive(out var message))
        {
            State.ProcessMessage(message);
            _currentAttempts = 0;
        }
        else
        {
            _currentAttempts++;

            if (_currentAttempts >= MaxAttempts)
            {
                throw new ConnectorException("Потеряно соединение с
сервером.");
            }
        }
    }
}

private bool TryReceive(out byte[] message)
{
    try
    {
        SetSocketTimeout();
        IPEndPoint endPoint = null;
        message = _udpClient.Receive(ref endPoint);
        return true;
    }
    catch (SocketException)
    {
        message = null;
        return false;
    }
    finally
    {
        ResetSocketTimeout();
    }
}

```

```

        private void SetSocketTimeout()
            => _udpClient.Client.ReceiveTimeout = ReceiveTimeout;

        private void ResetSocketTimeout()
            => _udpClient.Client.ReceiveTimeout = 0;
    }
}

```

Листинг В.7 – MatchConnectorStateBase.cs

```

using System.Threading.Tasks;

namespace Connectors.MatchConnectors
{
    internal abstract class MatchConnectorStateBase
    {
        protected MatchConnector Context { get; }

        public MatchConnectorStateBase(MatchConnector context)
            => Context = context;

        public abstract Task SendMessageAsync();
        public abstract void ProcessMessage(byte[] message);
    }
}

```

Листинг В.8 – HelloMatchState.cs

```

using System.Threading.Tasks;
using Network;
using Network.Messages;
using Newtonsoft.Json;

namespace Connectors.MatchConnectors
{
    internal class HelloMatchState : MatchConnectorStateBase
    {
        private readonly byte[] _message;

        public HelloMatchState(MatchConnector context) : base(context)
        {
            var privateEndPoint = new ClientEndPoint(Context.Ip,
Context.Port);
            var data = JsonConvert.SerializeObject(privateEndPoint);
            _message = MessageHelper.GetMessage(NetworkMessages.Hello, data);
        }

        public override async Task SendMessageAsync()
            => await Context.SendMessageAsync(_message);

        public override void ProcessMessage(byte[] message)
        {
            if (MessageHelper.GetMessageType(message) ==
NetworkMessages.Initial)
            {
                Context.State = new WaitMatchState(Context);
            }
        }
    }
}

```

Листинг В.9 – WaitMatchState.cs

```
using System;
using System.Threading.Tasks;
using Network;
using Network.Messages;
using Newtonsoft.Json;

namespace Connectors.MatchConnectors
{
    internal class WaitMatchState : MatchConnectorStateBase
    {
        private readonly byte[] _message;

        public WaitMatchState(MatchConnector context)
            : base(context)
        {
            => _message = MessageHelper.GetMessage(NetworkMessages.Hello);

            public override async Task SendMessageAsync()
            {
                => await Context.SendMessageAsync(_message);
            }

            public override void ProcessMessage(byte[] message)
            {
                if (MessageHelper.GetMessageType(message) ==
                    NetworkMessages.Initial)
                {
                    try
                    {
                        var data = MessageHelper.ToString(message);
                        Context.ConnectionMessage =
                            JsonConvert.DeserializeObject<ConnectionMessage>(data);
                        Context.FinishConnection();
                    }
                    catch (Exception ex)
                    {
                        if (ex is JsonReaderException || ex is
                            ArgumentException)
                        {
                            // TODO: лог о неправильном JSON'e?
                        }
                    }
                }
            }
        }
    }
}
```

Листинг В.10 – HolePuncher.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Threading.Tasks;
using Network;
using Network.Messages;
```

```

namespace Connectors.HolePuncher
{
    public class HolePuncher : IHolePuncher
    {
        private const int LoopDelay = 1000;

        private byte[] _packet;

        private UdpClient _udpClient;
        private uint _sessionId;
        private CancellationToken _cancellationToken;

        private List<ClientEndPoint> _potentials;
        private List<IPEndPoint> _connected;

        public async Task<List<IPEndPoint>> ConnectAsync(
            UdpClient udpClient,
            uint sessionId,
            IEnumerable<ClientEndPoint> clients,
            CancellationToken cancellationToken = default)
        {
            _udpClient = udpClient;
            _sessionId = sessionId;
            _cancellationToken = cancellationToken;
            _potentials = new List<ClientEndPoint>(clients);
            _connected = new List<IPEndPoint>();
            _packet = PacketHelper.CreatePacket(_sessionId, 0);

            await ConnectionLoop();
            return _connected;
        }

        private async Task ConnectionLoop()
        {
            while (_potentials.Any())
            {
                _cancellationToken.ThrowIfCancellationRequested();
                await Task.Delay(LoopDelay, _cancellationToken);
                await ConnectionFrame();
            }
        }

        private async Task ConnectionFrame()
        {
            await SendPackets();

            while (_udpClient.Available > 0)
            {
                try
                {
                    var result = await _udpClient.ReceiveAsync();
                    ProcessPacket(result.Buffer, result.RemoteEndPoint);
                }
                catch (SocketException e)
                {
                    Console.WriteLine(e.Message);
                }
            }
        }
    }
}

```

```

private async Task SendPackets()
{
    foreach (var client in _potentials)
    {
        await SendPacket(_packet, client.PublicEndPoint);

        if
(!IPAddress.IsLoopback(IPAddress.Parse(client.PublicEndPoint.Ip)))
        {
            await SendPacket(_packet, client.PrivateEndPoint);
        }
    }

    foreach (var client in _connected)
    {
        await SendPacket(_packet, client);
    }
}

private void ProcessPacket(byte[] message, IPEndPoint endPoint)
{
    if (message.Length >= PacketHelper.HeaderSize
        && PacketHelper.GetSessionId(message) == _sessionId
        && TryGetFromPotentials(endPoint, out var client))
    {
        _potentials.Remove(client);
        _connected.Add(endPoint);
    }
}

private async Task SendPacket(byte[] message, ClientEndPoint
endPoint)
    => await _udpClient.SendAsync(message, message.Length,
endPoint.Ip, endPoint.Port);

private async Task SendPacket(byte[] message, IPEndPoint endPoint)
    => await _udpClient.SendAsync(message, message.Length, endPoint);

private bool TryGetFromPotentials(IPEndPoint endPoint, out
ClientEndpoints client)
{
    client = _potentials.FirstOrDefault(
        c => c.PrivateEndPoint.IsSame(endPoint) ||
c.PublicEndPoint.IsSame(endPoint));
    return client != null;
}
}

```

Приложение Г
Исходный код библиотеки Network
(рекомендуемое)

Листинг Г.1 – ClientNetworkProxy.cs

```

using System;
using System.Linq;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Threading.Tasks;
using Network.NetworkBuffers;
namespace Network.Proxy
{
    public class ClientNetworkProxy : IClientNetworkProxy
    {
        private const int SendBufferSize = 10;
        private const int ReceiveBufferSize = 20;
        private const int Tolerance = 200;

        private readonly UdpClient _udpClient;

        private CancellationTokenSource _tokenSource;
        private ConcurrentNetworkBuffer _writeBuffer;
        private ConcurrentNetworkBuffer _readBuffer;

        private uint _sentPacketNumber;
        private uint _receivedPacketNumber;
        private uint[] _packetNumbersBuffer;
        private int _pointer;

        public uint SessionId { get; }
        public IPEndPoint Host { get; }
        public IWriteOnlyNetworkBuffer WriteBuffer => _writeBuffer;
        public IReadOnlyNetworkBuffer ReadBuffer => _readBuffer;

        public ClientNetworkProxy(UdpClient udpClient, uint sessionId,
IPPEndPoint host)
        {
            _udpClient = udpClient;
            SessionId = sessionId;
            Host = host;
        }

        public void Start()
        {
            _tokenSource = new CancellationTokenSource();
            _writeBuffer = new ConcurrentNetworkBuffer(SendBufferSize);
            _readBuffer = new ConcurrentNetworkBuffer(ReceiveBufferSize);
            _sentPacketNumber = 0;
            _receivedPacketNumber = 0;
            _pointer = 0;
            _packetNumbersBuffer = new uint[10];

            Task.Run(SendLoopAsync, _tokenSource.Token);
            Task.Run(ReceiveLoopAsync, _tokenSource.Token);
        }

        public void Stop()
        {
            _tokenSource?.Cancel();
            _tokenSource?.Dispose();
        }
    }
}

```

```

public void Dispose() => Stop();

private async Task SendLoopAsync()
{
    while (true)
    {
        _tokenSource.Token.ThrowIfCancellationRequested();
        await SendFrameAsync();
    }
}

private async Task SendFrameAsync()
{
    while (!_writeBuffer.IsEmpty)
    {
        var data = _writeBuffer.Read();
        var packet = PacketHelper.CreatePacket(SessionId,
        _sentPacketNumber++, data);
        await _udpClient.SendAsync(packet, packet.Length, Host);
    }
}

private async Task ReceiveLoopAsync()
{
    while (true)
    {
        try
        {
            _tokenSource.Token.ThrowIfCancellationRequested();
            await ReceiveFrameAsync();
        }
        catch (SocketException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}

private async Task ReceiveFrameAsync()
{
    if (_udpClient.Available > 0)
    {
        var result = await _udpClient.ReceiveAsync();

        if (Equals(result.RemoteEndPoint, Host))
        {
            if (PacketHelper.GetSessionId(result.Buffer) !=
SessionId) return;
            var packetNumber = PacketHelper.GetNumber(result.Buffer);
            _packetNumbersBuffer[_pointer++] = packetNumber;

            if (_pointer == _packetNumbersBuffer.Length)
            {
                _pointer = 0;

                if
(PacketHelper.IsShouldCorrectPacketNumber(_packetNumbersBuffer, Tolerance))
                {
                    _receivedPacketNumber =
_packetNumbersBuffer.Min();
                }
            }
        }
    }
}

```



```

        _packetsInfo = new Dictionary<IPEndPoint, (uint MaxNumber, uint[]
Buffer, int Pointer)>();

        foreach (var client in Clients)
        {
            _readBuffers[client] = new
ConcurrentNetworkBuffer(ReceiveBufferSize);
            _writeBuffers[client] = new
ConcurrentNetworkBuffer(SendBufferSize);
            _packetsInfo[client] = (0, new uint[10], 0);
        }

        Task.Run(SendLoopAsync);
        Task.Run(ReceiveLoopAsync);
    }

    public void Stop()
    {
        _tokenSource?.Cancel();
        _tokenSource?.Dispose();
    }

    public IReadOnlyNetworkBuffer GetReadBuffer(IPEndPoint endPoint) =>
_readBuffers[endPoint];

    public IWriteOnlyNetworkBuffer GetWriteBuffer(IPEndPoint endPoint) =>
_writeBuffers[endPoint];

    private async Task SendLoopAsync()
    {
        while (true)
        {
            _tokenSource.Token.ThrowIfCancellationRequested();
            await SendFrameAsync();
        }
    }

    protected virtual async Task SendFrameAsync()
    {
        foreach (var clientBufferPair in _writeBuffers)
        {
            var ip = clientBufferPair.Key;
            var buffer = clientBufferPair.Value;

            while (!buffer.IsEmpty)
            {
                var data = buffer.Read();
                var packet = PacketHelper.CreatePacket(SessionId,
_sentPacketNumber, data);
                await _udpClient.SendAsync(packet, packet.Length, ip);
            }

            _sentPacketNumber++;
        }
    }

    private async Task ReceiveLoopAsync()
    {
        while (true)
        {
            try
            {
                _tokenSource.Token.ThrowIfCancellationRequested();
            }

```

```

        await ReceiveFrameAsync();
    }
    catch (SocketException e)
    {
        Console.WriteLine(e.Message);
    }
}

private async Task ReceiveFrameAsync()
{
    if (_udpClient.Available > 0)
    {
        var result = await _udpClient.ReceiveAsync();

        if (_readBuffers.TryGetValue(result.RemoteEndPoint, out var
buffer))
        {
            if (PacketHelper.GetSessionId(result.Buffer) !=
SessionId) return;
            var packetNumber = PacketHelper.GetNumber(result.Buffer);
            var packetInfo = _packetsInfo[result.RemoteEndPoint];
            packetInfo.Buffer[packetInfo.Pointer++] = packetNumber;

            if (packetInfo.Pointer == packetInfo.Buffer.Length)
            {
                packetInfo.Pointer = 0;

                if
(PacketHelper.IsShouldCorrectPacketNumber(packetInfo.Buffer, Tolerance))
                {
                    packetInfo.MaxNumber = packetInfo.Buffer.Min();
                }
            }

            if (packetNumber >= packetInfo.MaxNumber)
            {
                buffer.Write(PacketHelper.GetData(result.Buffer));
                packetInfo.MaxNumber = packetNumber;
            }
        }
    }
}
}
}
}

```

Приложение Д

Листинг клиентских классов, ответственных за сетевое взаимодействие
(рекомендуемое)

Листинг Д.1 – Reconcilator.cs

```

using System.Collections.Generic;
using System.Linq;
using Assets.Scripts.ECS.Components;
using Assets.Scripts.Networking.NetworkingManagers;
using Assets.Scripts.Networking.Serializers;
using Assets.Scripts.Util;
using ECS.Core;
using Network.Proxy;
using UnityEngine;

public class Reconcilator : IEngineWrapper
{
    private readonly Engine _engine;
    private readonly ClientNetworkProxy _clientProxy;

    public Reconcilator(Engine engine, ClientNetworkProxy clientProxy)
    {
        _engine = engine;
        _clientProxy = clientProxy;
    }

    public void Update(double dt)
    {
        Send();
        Receive();
        _engine.Update(dt);
    }

    private void Receive()
    {
        if (!_clientProxy.ReadBuffer.IsEmpty)
        {
            var message = _clientProxy.ReadBuffer.ReadLast();

            try
            {
                var snapshot =
                    MessageHelper.GetSnapshot(EcsContextHelper.HostWorldContext, message);
                var reconcilableEntities = _engine.GetIdentifiedEntities();

                foreach (var entity in snapshot)
                {
                    var reconcilable = reconcilableEntities[entity.Id()];

                    foreach (var typeComponentPair in entity.ToList())
                    {
                        reconcilable.Remove(typeComponentPair.Key);
                        reconcilable.Add(typeComponentPair.Value);
                    }
                }
            }
            catch
            {
                Debug.Log("Не удалось десериализовать сообщение.");
            }
        }
    }
}

```

```

private void Send()
{
    var serializableEntities = GetSerializableEntities();
    var message =
MessageHelper.GetMessage(EcsContextHelper.ClientWorldContext,
serializableEntities);
    _clientProxy.WriteBuffer.Write(message);
}

private List<Entity> GetSerializableEntities()
    => _engine.GetEntities().Filter<IsSerializableComponent>().ToList();
}

```

Листинг Д.2 – Snapshoter.cs

```

using System.Collections.Generic;
using System.Linq;
using Assets.Scripts.ECS.Components;
using Assets.Scripts.Networking.NetworkingManagers;
using Assets.Scripts.Networking.Serializers;
using Assets.Scripts.Util;
using ECS.Core;
using Network.Proxy;
using UnityEngine;

public class Snapshoter : IEngineWrapper
{
    private readonly Engine _engine;
    private readonly HostNetworkProxy _hostProxy;

    public Snapshoter(Engine engine, HostNetworkProxy hostProxy)
    {
        _engine = engine;
        _hostProxy = hostProxy;
    }

    public void Update(double dt)
    {
        Receive();
        _engine.Update(dt);
        Send();
    }

    private void Receive()
    {
        var receivedSnapshots = new List<List<Entity>>>();

        foreach (var client in _hostProxy.Clients)
        {
            var buffer = _hostProxy.GetReadBuffer(client);

```

```

        if (!buffer.IsEmpty)
        {
            try
            {
                var message = buffer.ReadLast();
                var snapshot =
MessageHelper.GetSnapshot(EcsContextHelper.ClientWorldContext, message);
                receivedSnapshots.Add(snapshot);
            }
            catch
            {
                Debug.Log("Не удалось десериализовать сообщение.");
            }
        }
    }

    var identifiedEntities = _engine.GetIdentifiedEntities();

    foreach (var snapshot in receivedSnapshots)
    {
        foreach (var entity in snapshot)
        {
            var entityToUpdate = identifiedEntities[entity.Id()];

            foreach (var typeComponentPair in entity.ToList())
            {
                entityToUpdate.Remove(typeComponentPair.Key);
                entityToUpdate.Add(typeComponentPair.Value);
            }
        }
    }

    private void Send()
    {
        var serializableEntities = GetSerializableEntities();
        var message =
MessageHelper.GetMessage(EcsContextHelper.HostWorldContext,
serializableEntities);

        foreach (var client in _hostProxy.Clients)
        {
            var buffer = _hostProxy.GetWriteBuffer(client);
            buffer.Write(message);
        }
    }

    private List<Entity> GetSerializableEntities()
    => _engine.GetEntities().Filter<IsSerializableComponent>().ToList();
}

```

Приложение Е
Результаты функционального тестирования веб-API
(рекомендуемое)

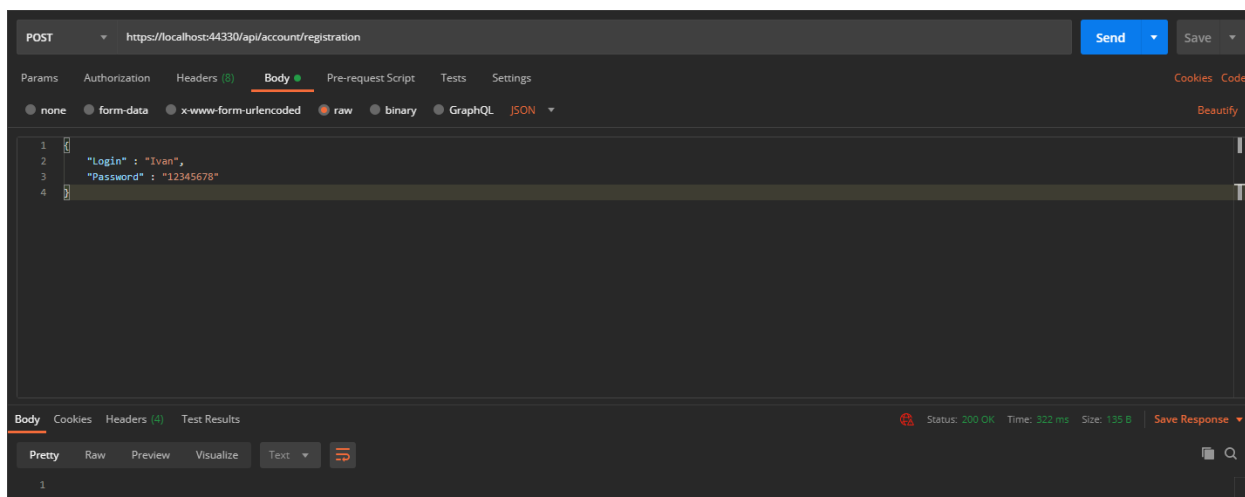


Рисунок Е.1 – Регистрация нового пользователя с корректным, не занятым логином и корректным паролем

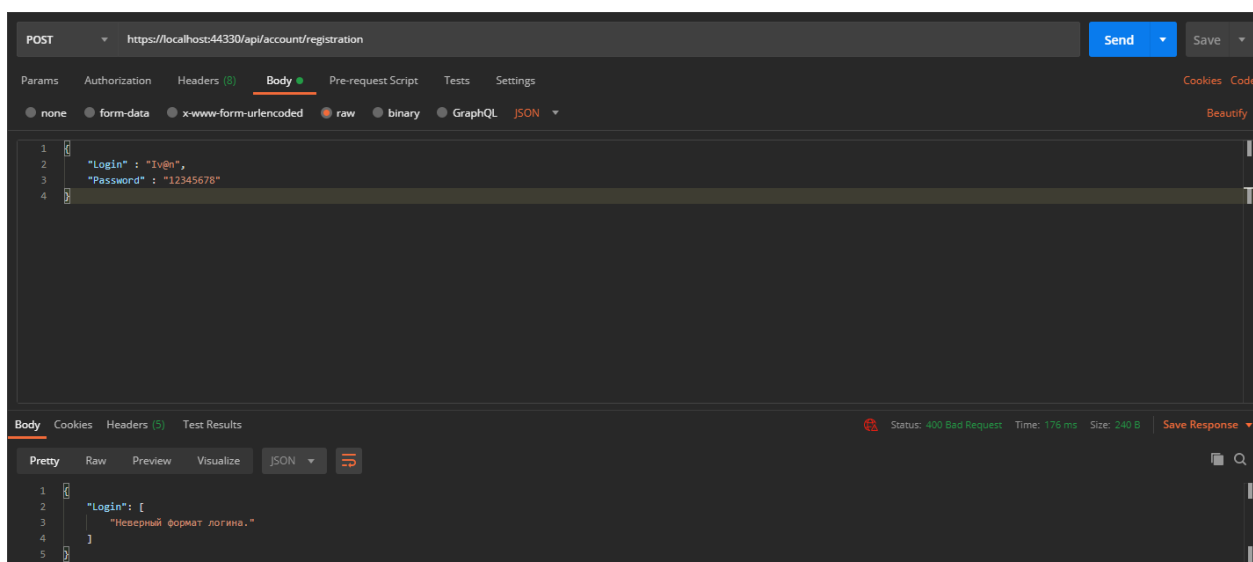


Рисунок Е.2 – Регистрация нового пользователя с некорректным логином

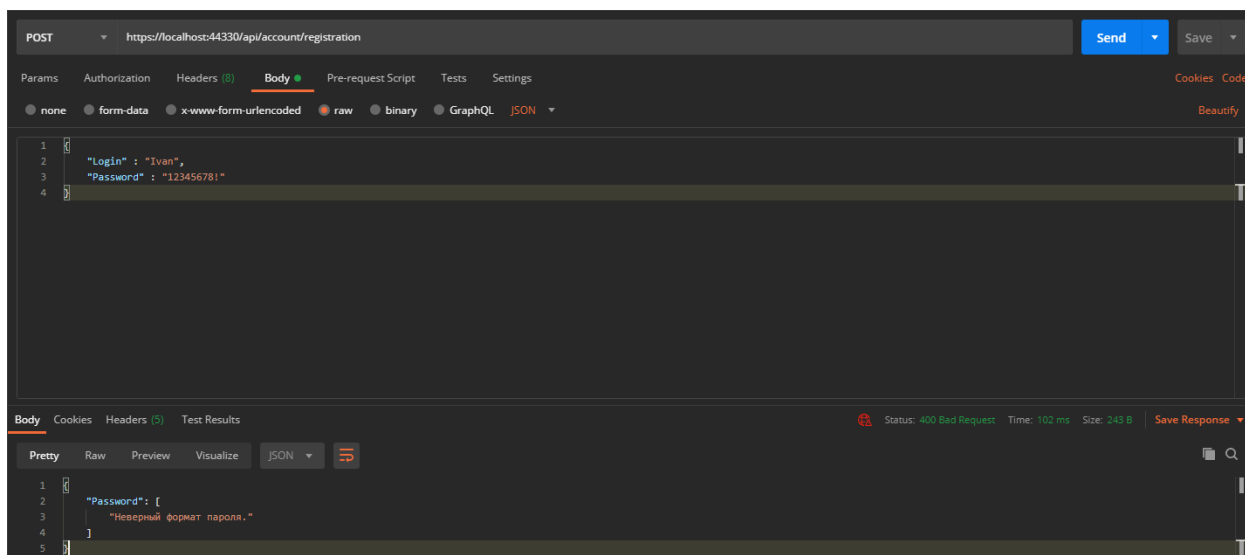


Рисунок Е.3 – Регистрация нового пользователя с некорректным паролем

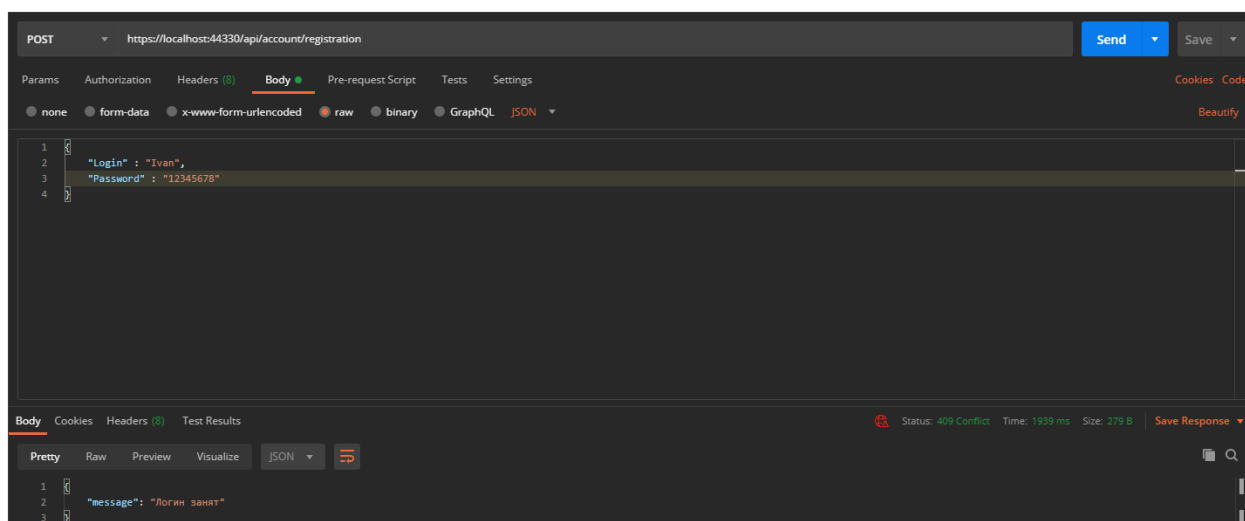


Рисунок Е.4 – Регистрация пользователя под занятым паролем

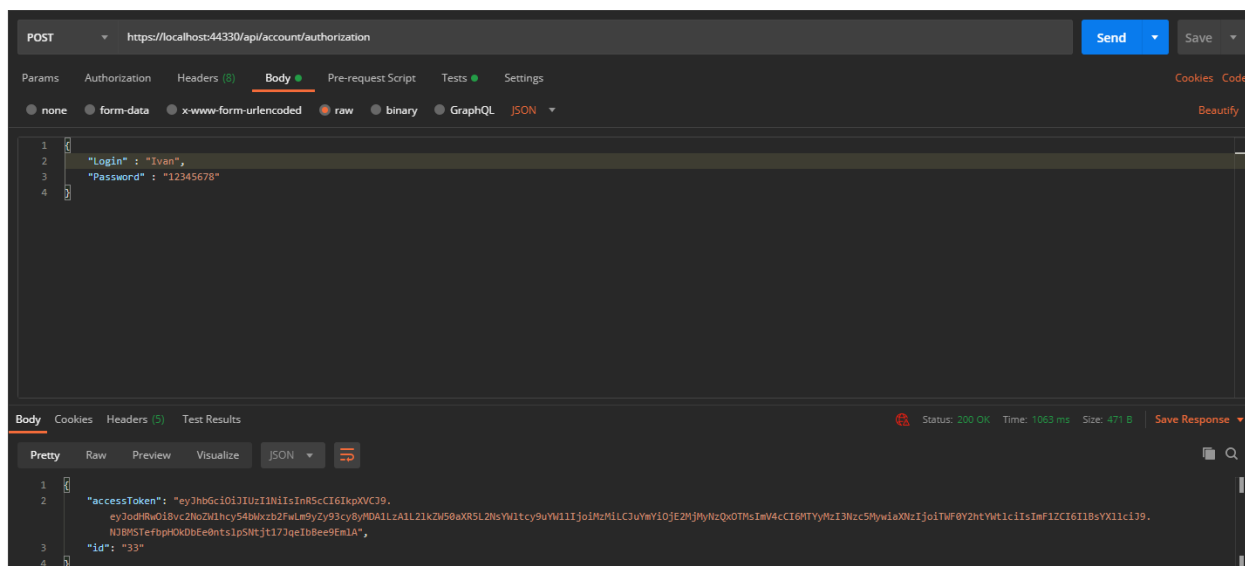


Рисунок Е.5 – Вход в систему с использованием логина и пароля существующего пользователя

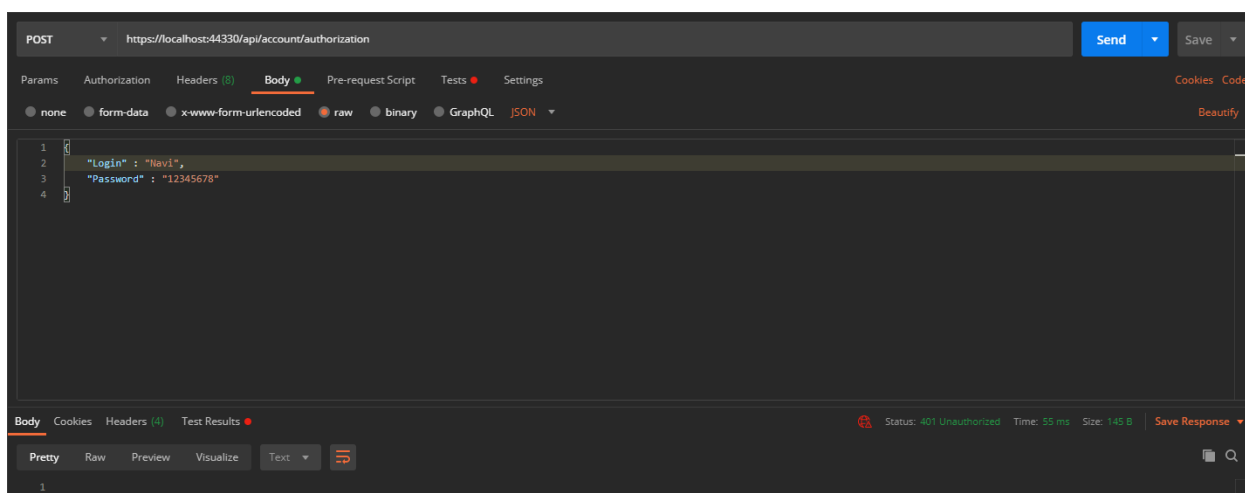


Рисунок Е.6 – Вход в систему под несуществующим логином

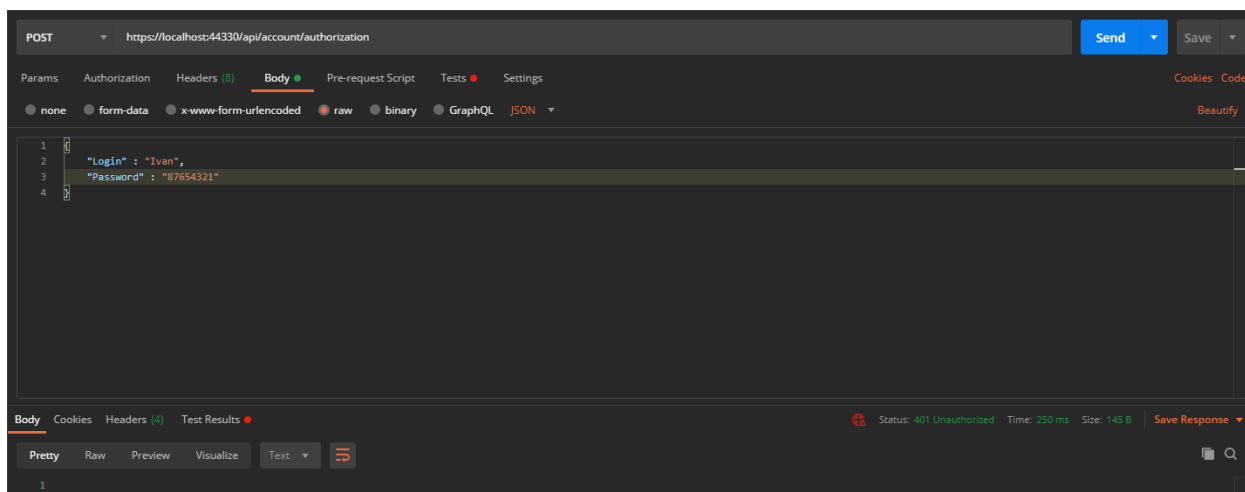


Рисунок Е.7 – Вход в систему с неверным паролем

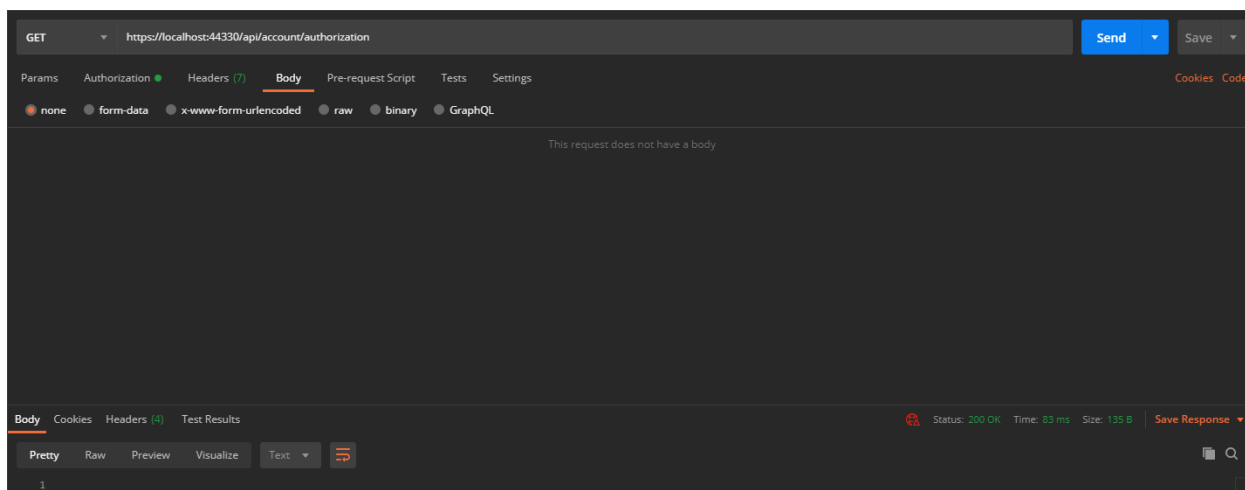


Рисунок Е.8 – Проверка, вошел ли пользователь в систему с активной сессией

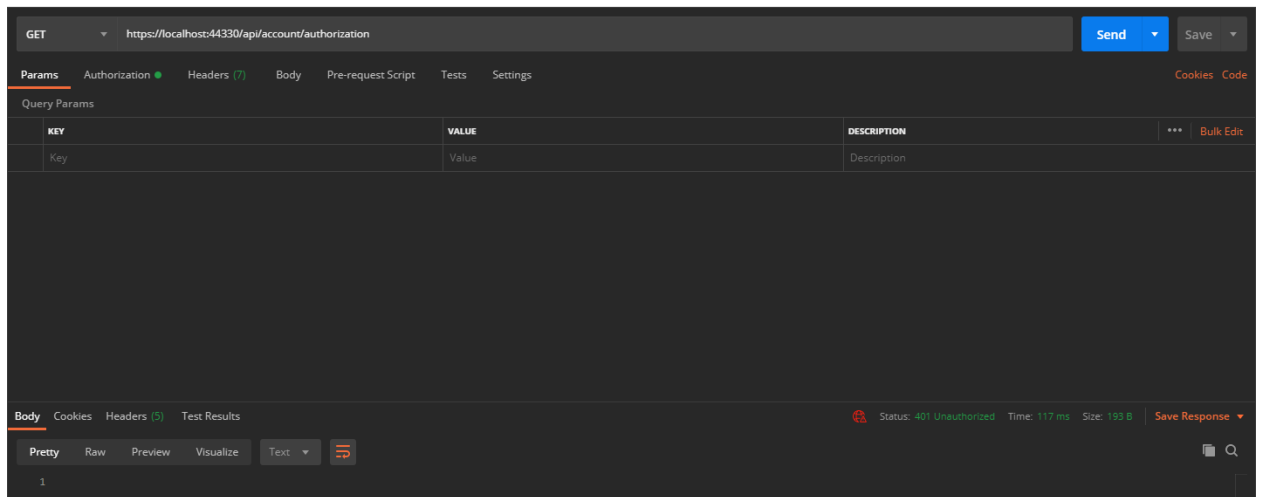


Рисунок Е.9 – Проверка, вошел ли пользователь в систему с неактивной сессией

Приложение Ж

Результаты функционального тестирования клиентского приложения
(рекомендуемое)



Рисунок Ж.1 – Регистрация пользователя с корректным, не занятым логином и корректным паролем

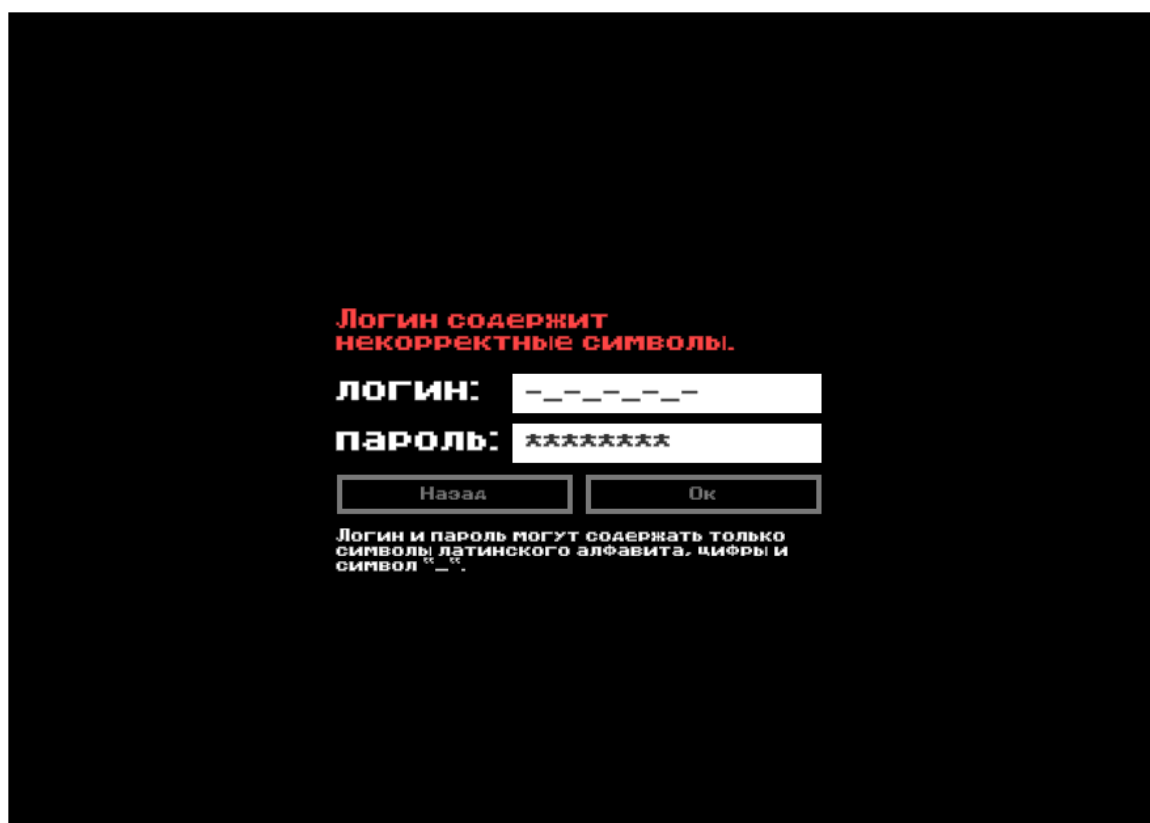


Рисунок Ж.2 – Регистрация пользователя с некорректным логином

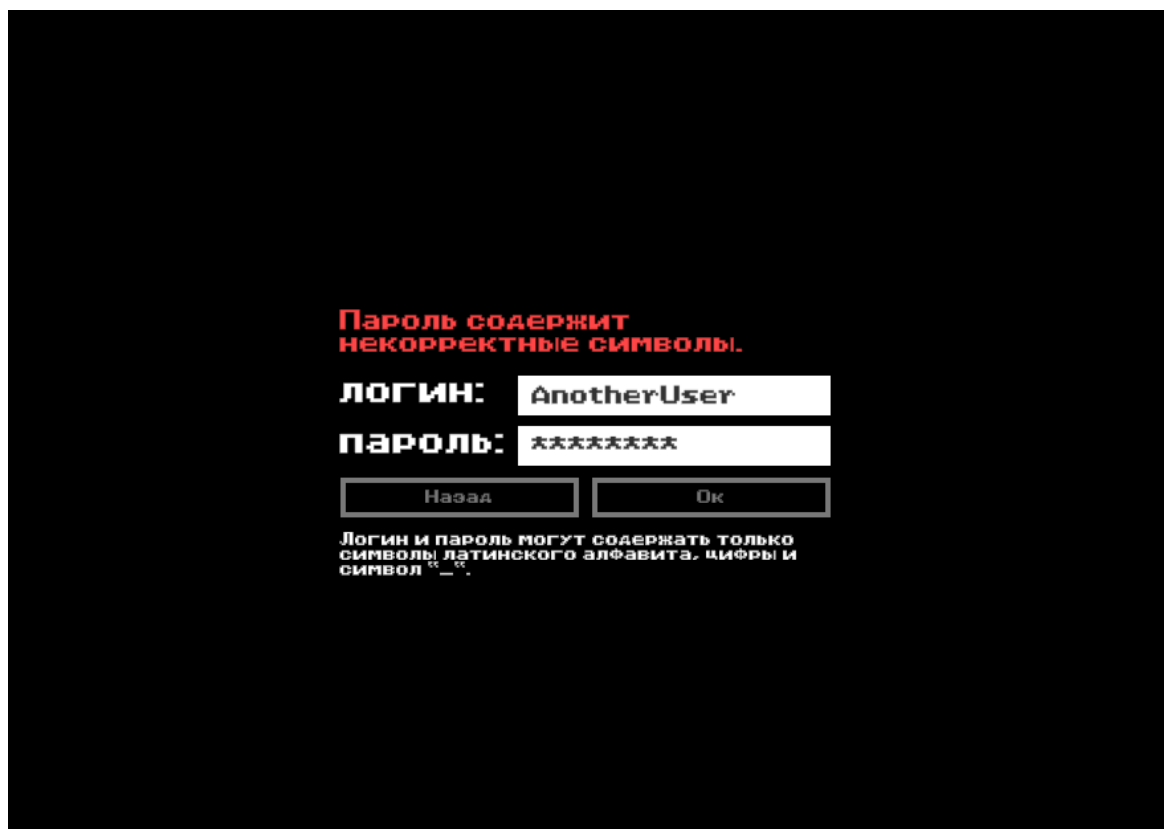


Рисунок Ж.3 – Регистрация пользователя с некорректным паролем

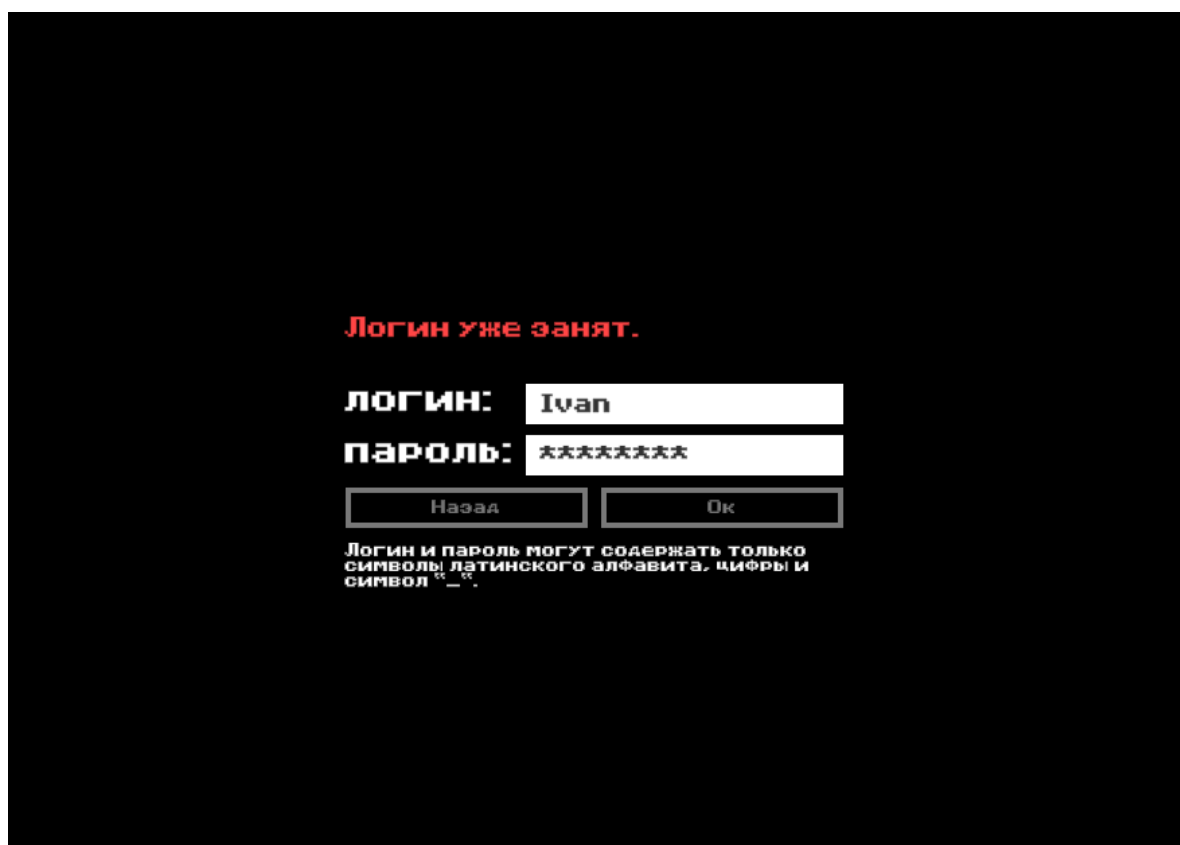


Рисунок Ж.4 – Регистрация пользователя с занятым логином

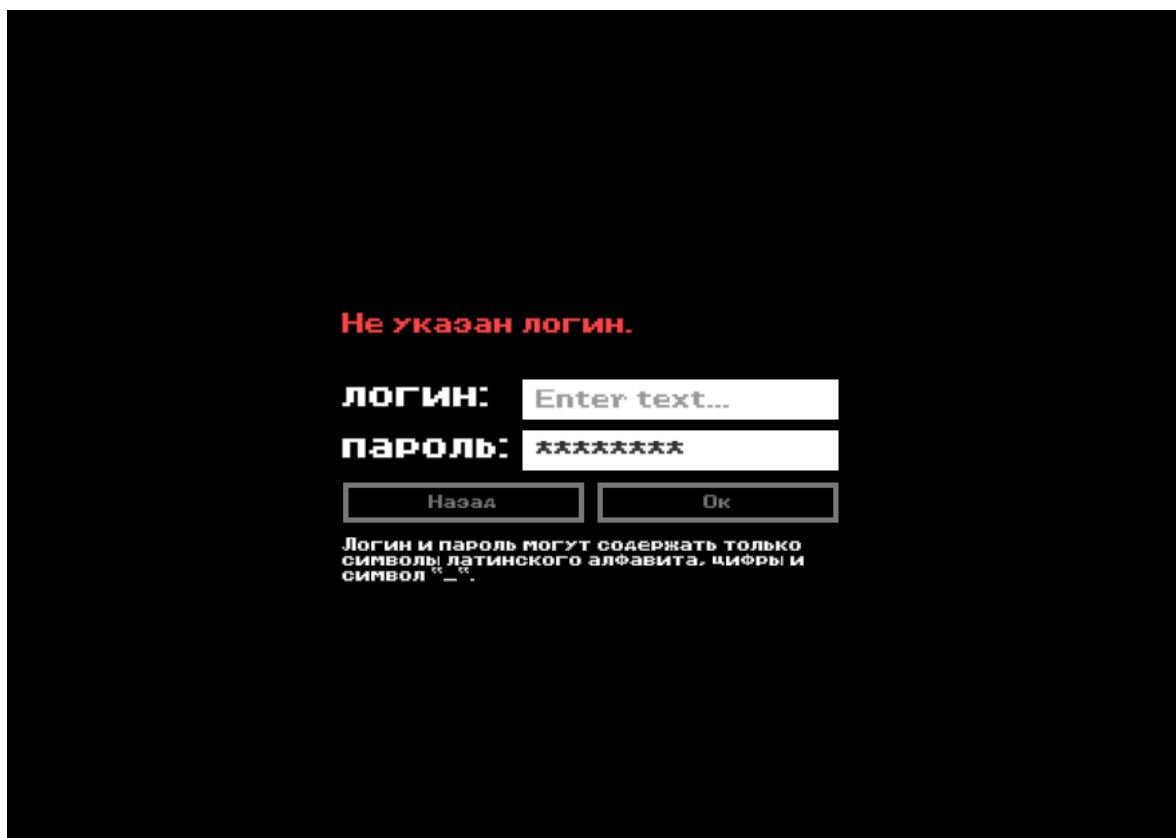


Рисунок Ж.5 – Регистрация пользователя без логина

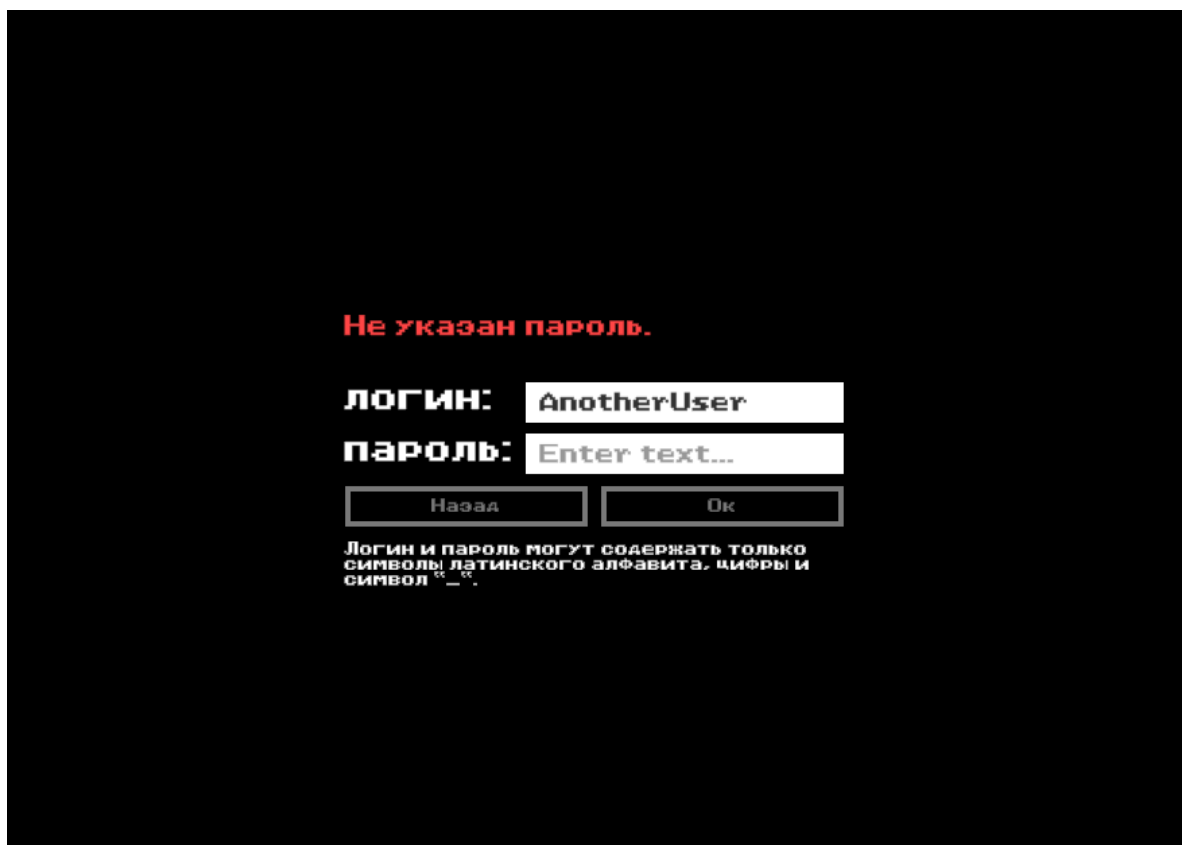


Рисунок Ж.6 – Регистрация пользователя без пароля

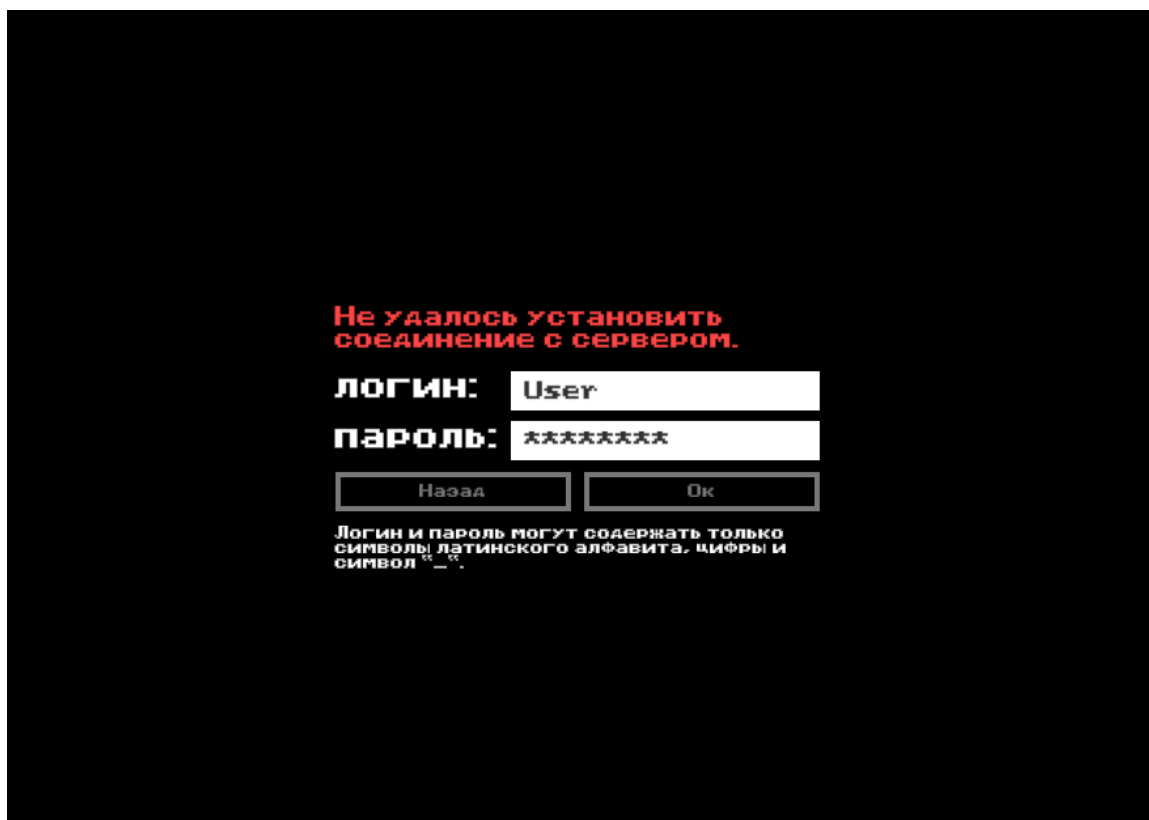


Рисунок Ж.7 – Попытка регистрации при незапущенном сервере



Рисунок Ж.8 – Вход в систему с использованием логина и пароля
существующего пользователя

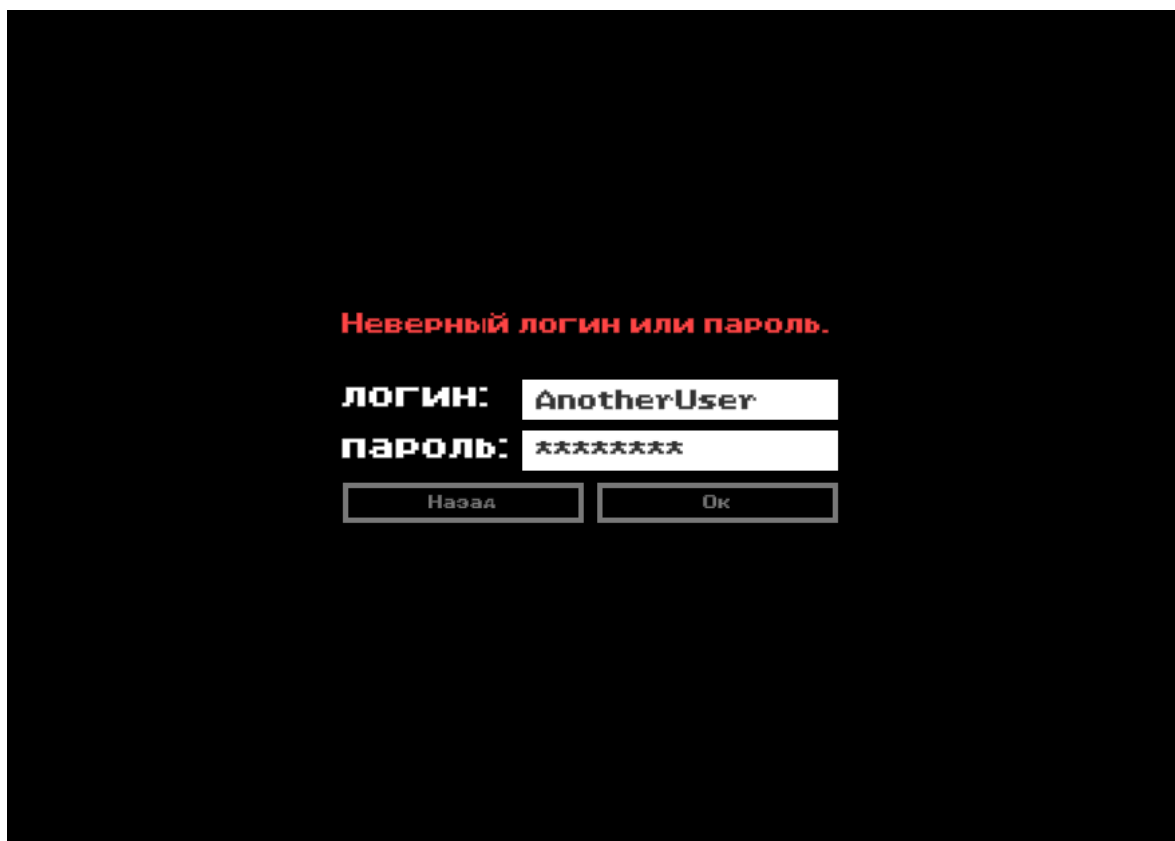


Рисунок Ж.9 – Вход в систему под несуществующим логином

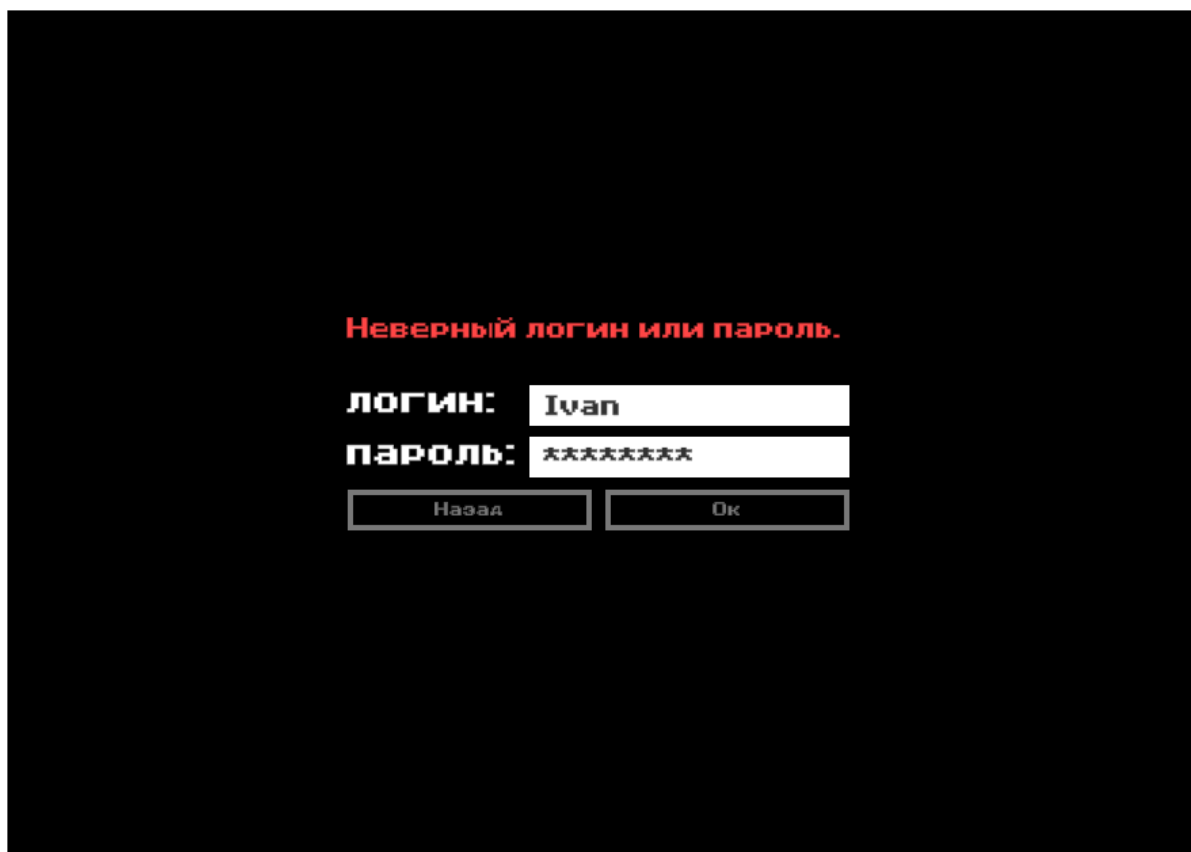


Рисунок Ж.10 – Вход в систему с неверным паролем

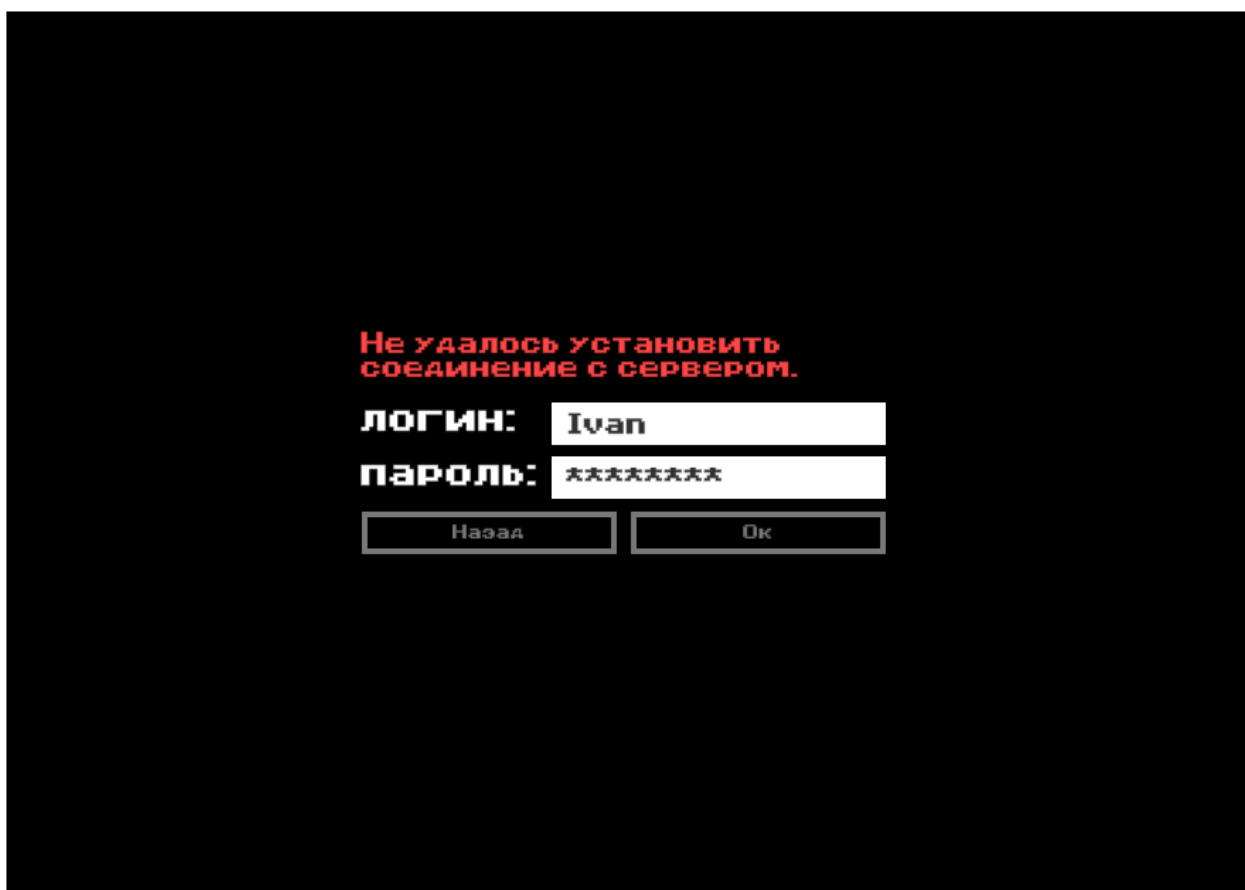


Рисунок Ж.11 – Вход в систему с незапущенным сервером

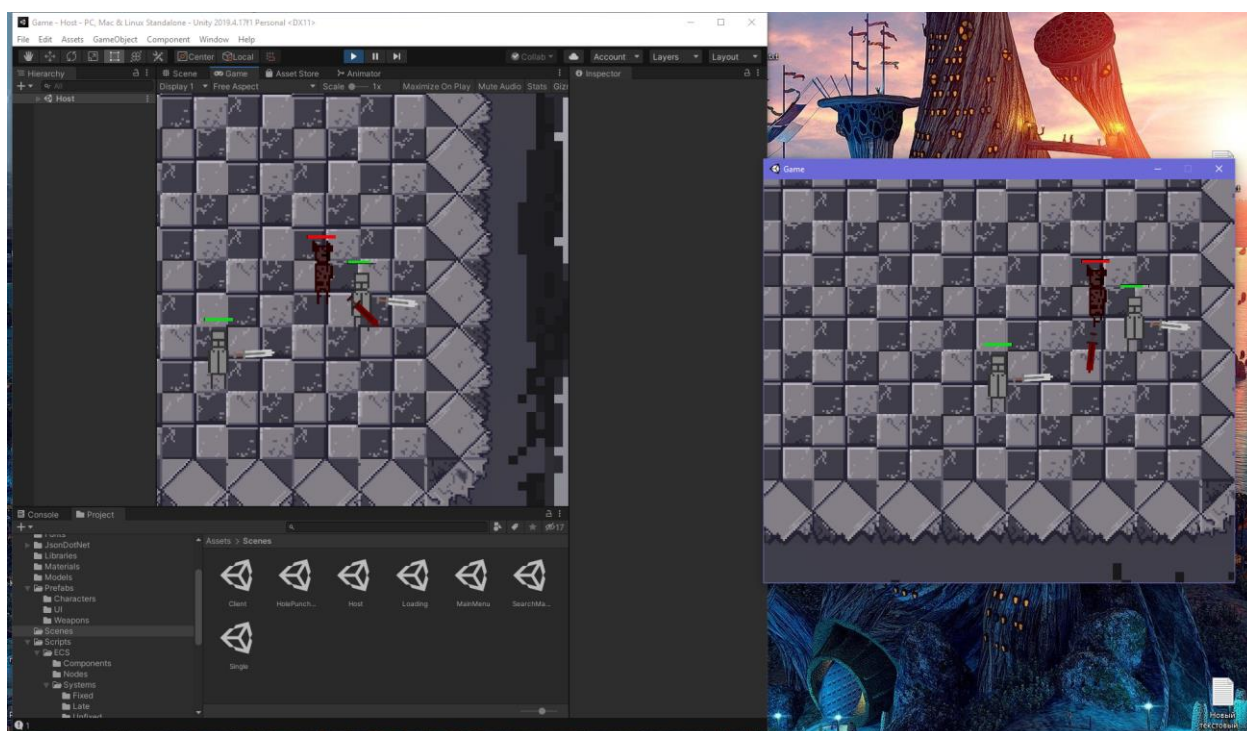


Рисунок Ж.12 – Два пользователя в одном матче

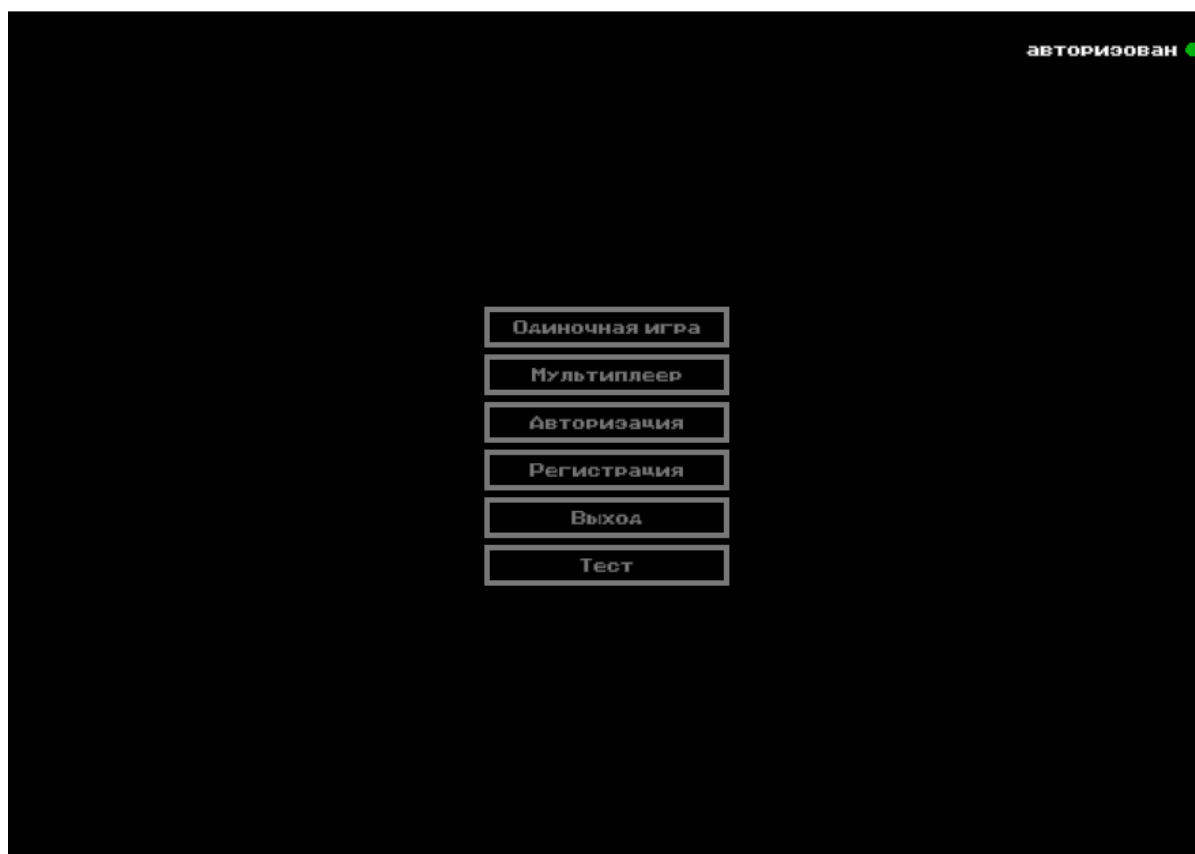


Рисунок Ж.13 – Возврат в главное меню после остановки поиска матча

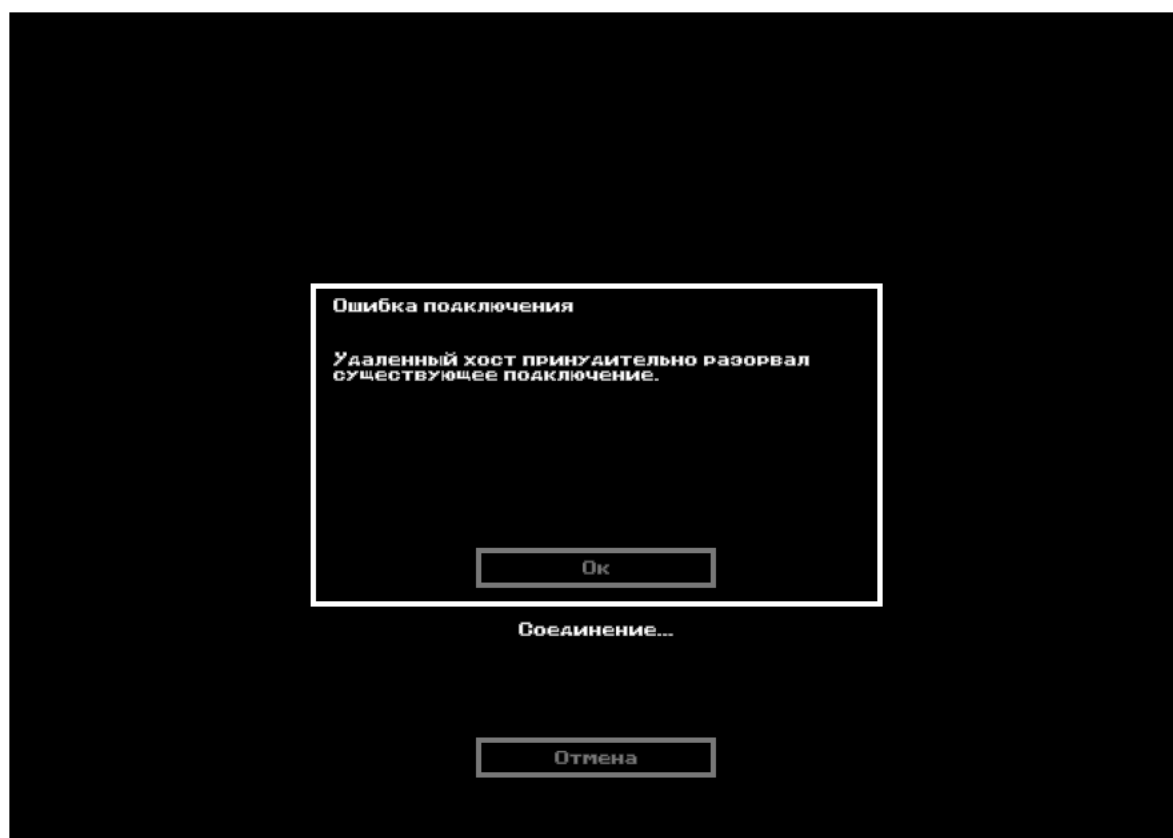


Рисунок Ж.14 – Один из пользователей прервал соединение



Рисунок Ж.15 – Попытка дважды инициировать поиск матча одним и тем же пользователем

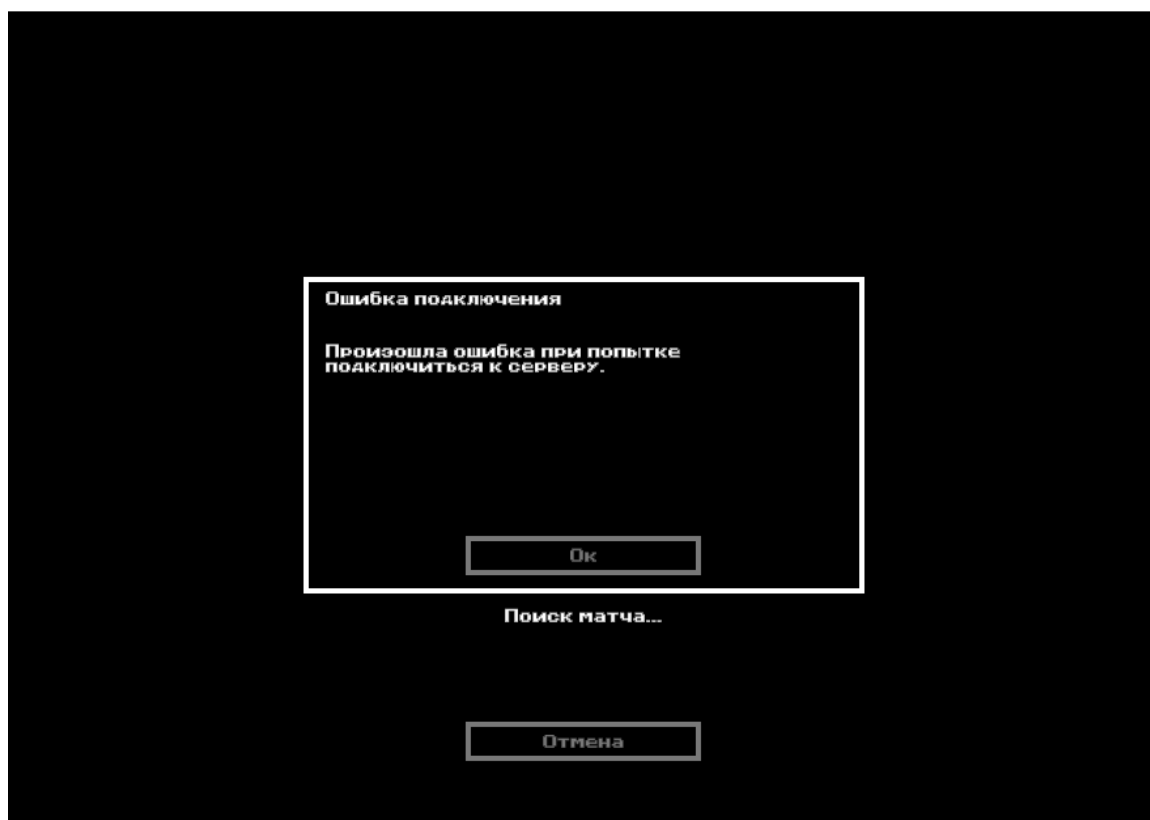


Рисунок Ж.16 – Попытка поиска матча с незапущенным сервером

Приложение И
Результаты исследовательского тестирования игры
(рекомендуемое)

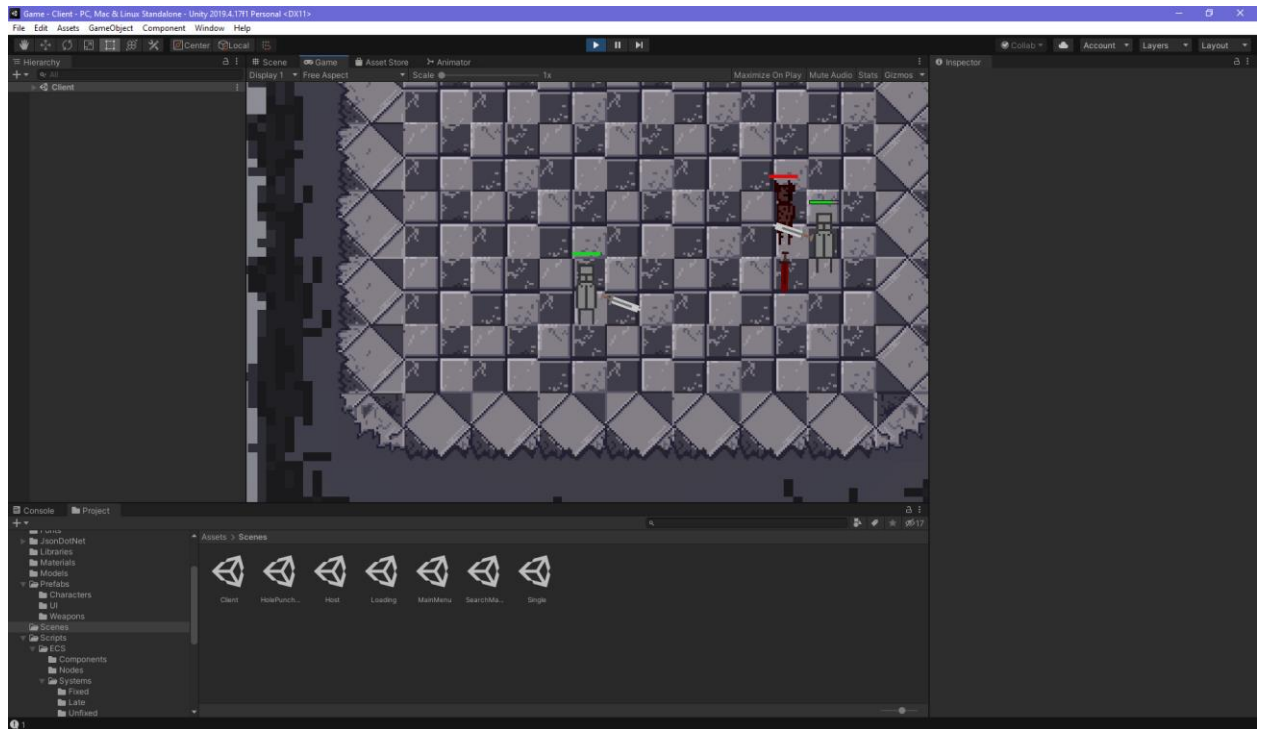


Рисунок И.1 – Персонаж до перемещения

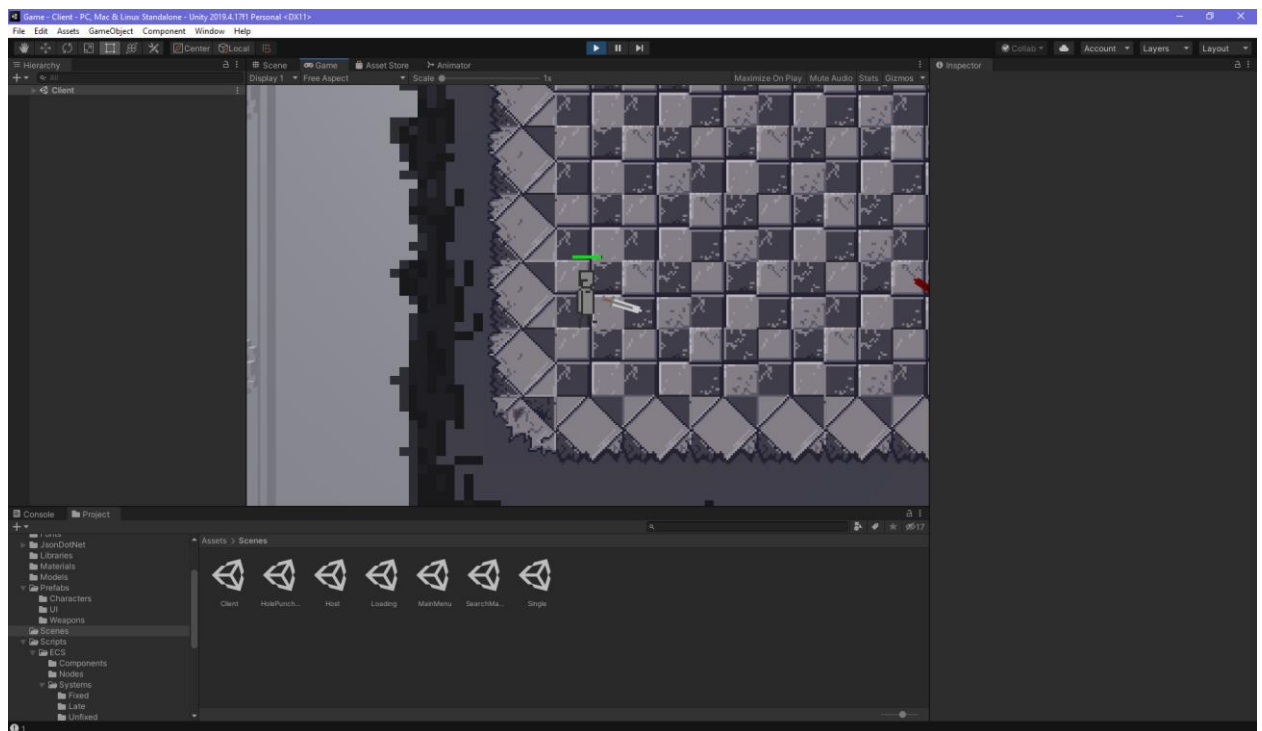


Рисунок И.2 Персонаж после перемещения

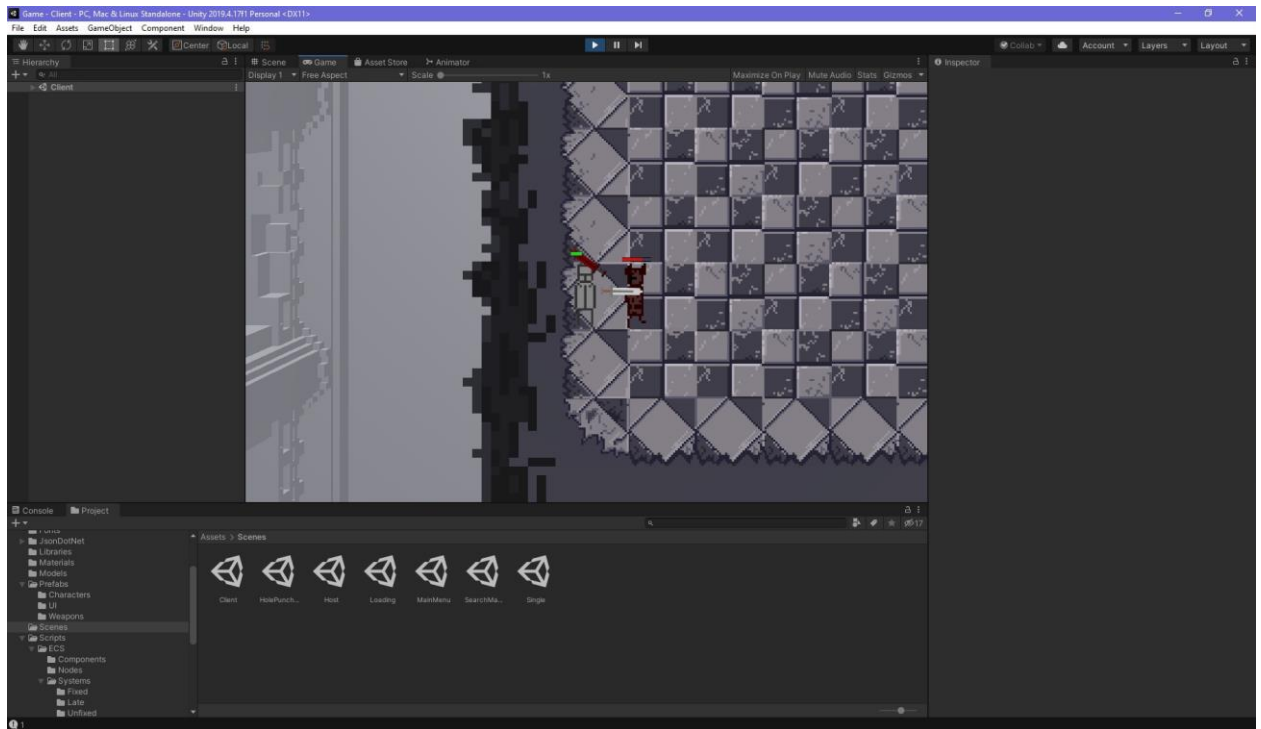


Рисунок И.3 – Здоровье противника до нанесения урона

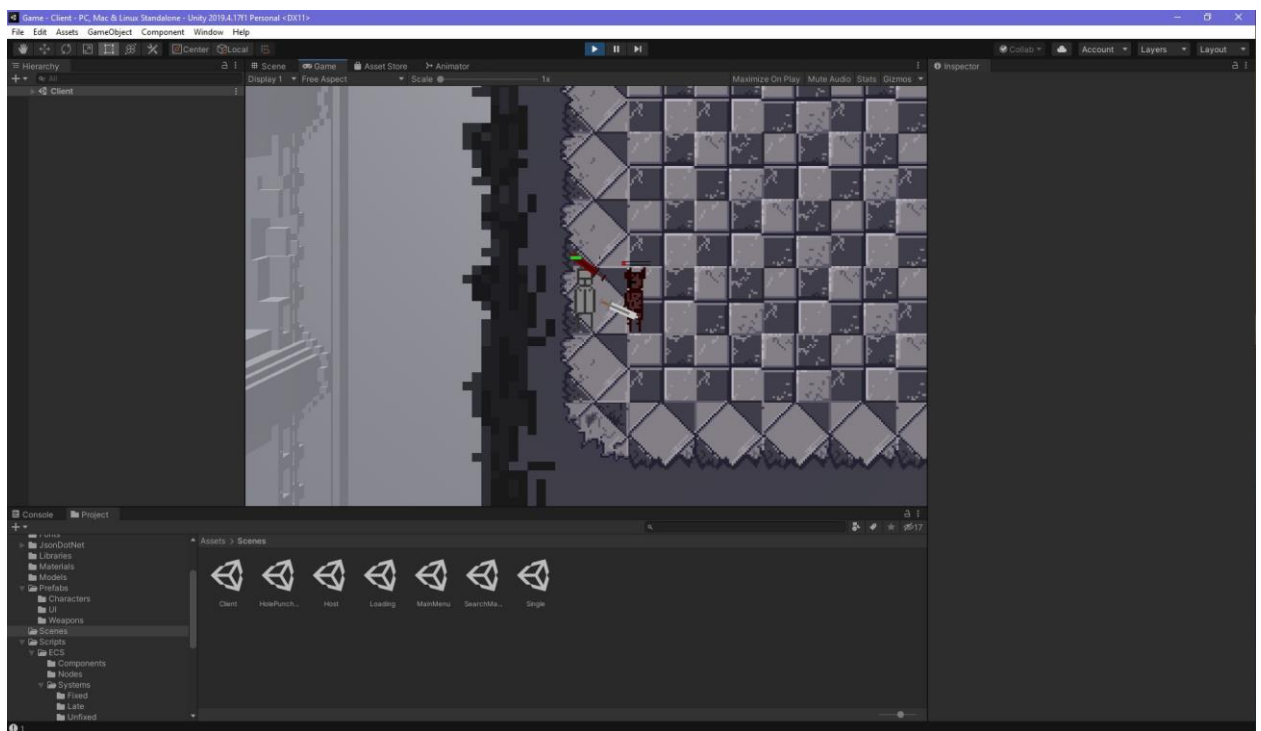


Рисунок И.4 – Здоровье противника после нанесения урона

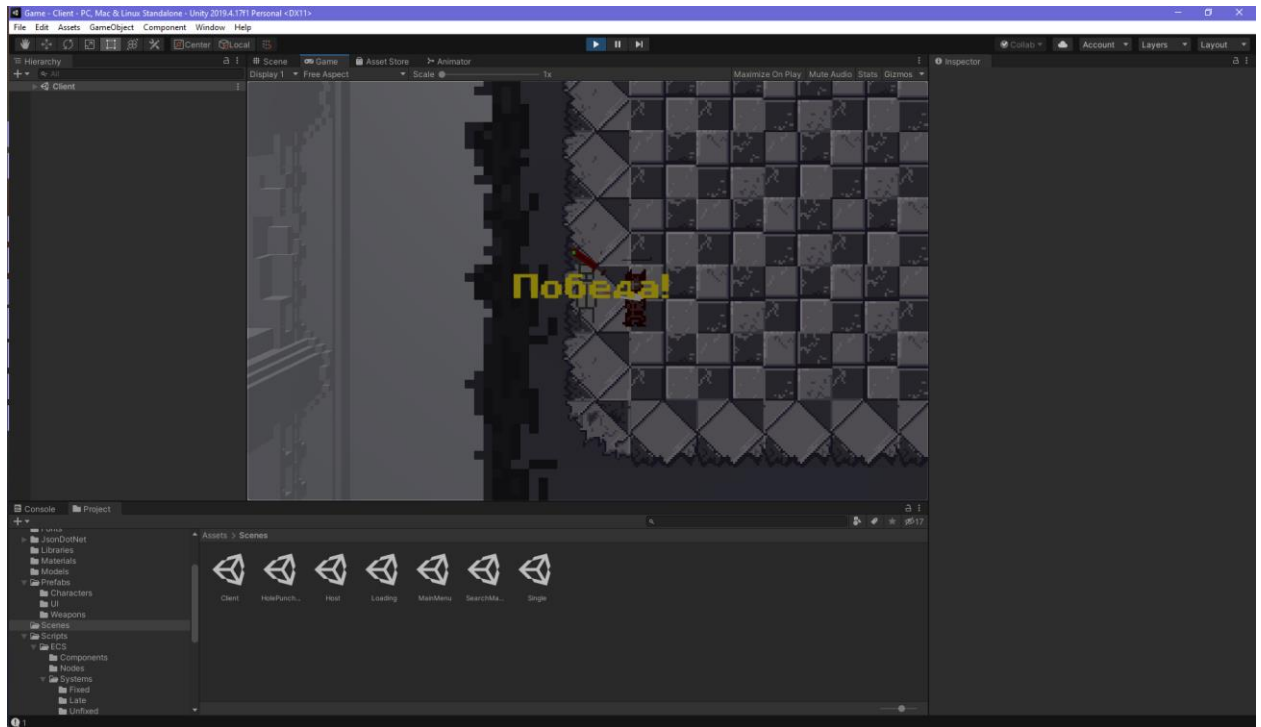


Рисунок И.5 – Победа

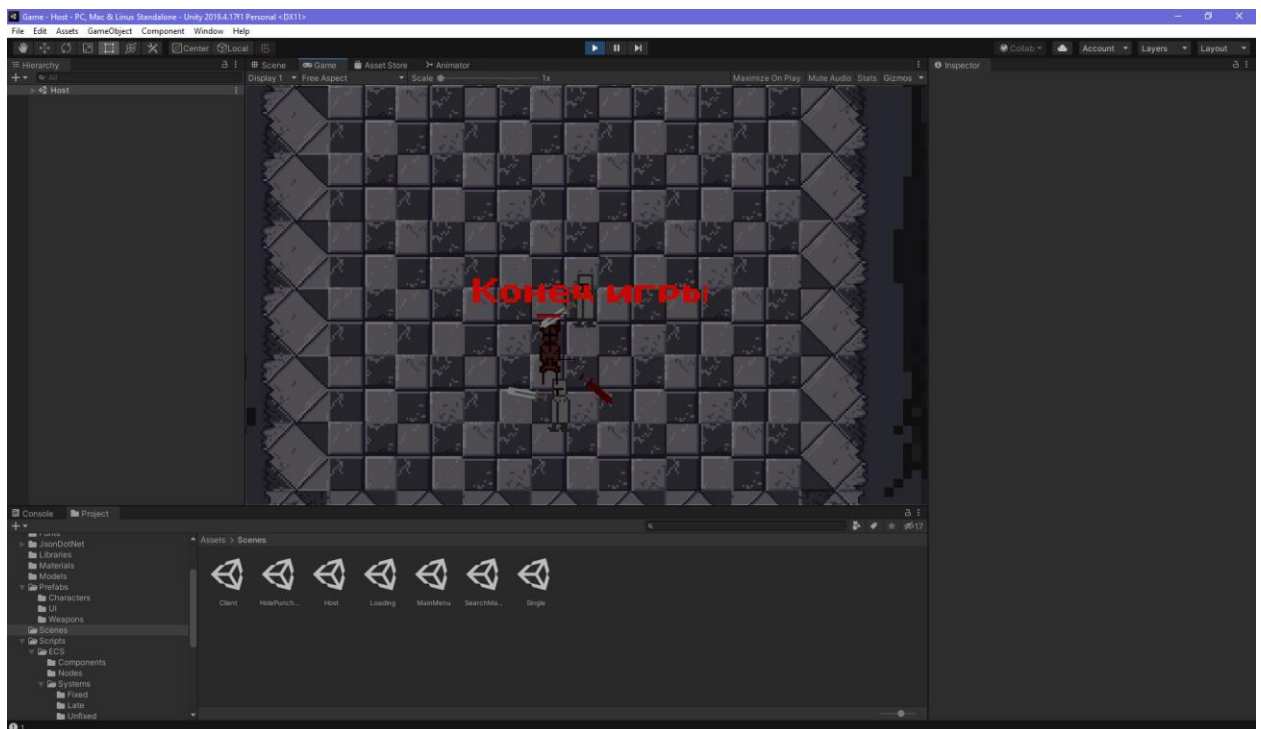


Рисунок И.6 - Поражение