



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное автономное образовательное учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

---

ФАКУЛЬТЕТ «Робототехника и комплексная автоматизация»

КАФЕДРА «Системы автоматизированного проектирования (РК-6)»

## РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

**НА ТЕМУ:**

**«Разработка сетевых методов автоматизированного  
запуска распределённой системы выделенных серверов  
Unreal Engine 4»**

Студент РК6-41М  
(Группа)

\_\_\_\_\_  
(Подпись, дата)

**Боженко Д.В.**  
(Фамилия И.О.)

Руководитель ВКР

\_\_\_\_\_  
(Подпись, дата)

**Витюков Ф.А.**  
(Фамилия И.О.)

Нормоконтролёр

\_\_\_\_\_  
(Подпись, дата)

**Грошев С.В.**  
(Фамилия И.О.)

2025 г.

## АННОТАЦИЯ

В представленной работе описан поэтапный процесс разработки распределенной системы автоматизированного запуска выделенных серверов Unreal Engine 4. Работа содержит в себе подробное техническое описание каждого этапа разработки системы и описание проведенного исследования в области решаемой задачи и обоснования выбранного решения. В работе описаны и проиллюстрированы следующие этапы реализации проекта: разработка программы менеджера серверов; разработка программы-демона; реализация механизма авторизации для контроля доступа к запуску выделенного сервера; реализация механизма хранения информации о транзакциях на стороне клиента; реализация системы мониторинга нагрузки приложения через консольный интерфейс. Расчетно-пояснительная записка содержит 70 с., 22 рис., 13 источников, 1 прил.

Тип работы: выпускная квалификационная работа.

Тема работы: «Разработка сетевых методов автоматизированного запуска распределённой системы выделенных серверов Unreal Engine 4».

Объекты исследований: разработка сетевых методов, разработка распределенной системы, развертывание и запуск выделенных серверов Unreal Engine.

## ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

*Unreal Engine 4 (UE)* – движок Unreal Engine 4, разрабатываемый и поддерживаемый компанией Epic Games.

*Выделенный сервер Unreal Engine 4 (сервер)* – программа на Unreal Engine 4, которая не имеет графического интерфейса и механизмов обработки 3D-графики и служит для обеспечения взаимодействия пользователей в едином виртуальном пространстве.

*Контейнер* – это изолированное пространство, которое позволяет запускать приложения с их зависимостями отдельно от основной системы, но которое одновременно использует ядро основной системы.

*Docker* – это программное обеспечение для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации, контейнеризатор приложений.

*TCP* – протокол транспортного уровня, обеспечивающий надежную доставку данных.

*Epic Online Services (EOS)* – встроенная в движок подсистема, предоставляющая низкоуровневые программные инструменты.

*Клиент* – программа на Unreal Engine, которая имеет пользовательский графический интерфейс и отправляет команды на желание начать игровую сессию.

*Менеджер серверов (менеджер)* – программа, находящаяся на отдельном от клиента узле, которая принимает входящие команды от клиентов и запущенных экземпляров выделенного сервера UE.

*Программа-демон (демон)* – программа, находящаяся на одном узле с запускаемыми экземплярами выделенных серверов UE и управляющая их процессами в операционной системе.

*Command-line interface (CLI)* – это интерфейс командной строки, с помощью которого пользователь может взаимодействовать с программным обеспечением и операционной системой путем ввода команд.

*Graphical user interface (GUI)* – это графический интерфейс, с помощью которого пользователь может взаимодействовать с программным обеспечением и операционной системой. Как правило, к компонентам графического интерфейса относятся окна, кнопки, выпадающие списки, слайдеры и другие графические элементы.

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ .....	10
1    Запуск приложения с помощью технологии контейнеризации .....	12
2    Разработка собственных сетевых сервисов системы .....	15
2.1    Разработка менеджера на Unreal Engine .....	17
2.2    Разработка менеджера на C++ и Boost.Asio .....	19
2.3    Разработка демона на C++ и Boost.Asio .....	23
2.4    Программная реализация логгера .....	30
2.5    Запуск сервисов на операционной системе Linux Debian .....	32
3    Реализация механизма авторизации для контроля запуска серверов .....	38
3.1    Программная реализация авторизации .....	39
3.2    Реализация хранения информации о транзакциях на стороне клиента .....	41
4    Обеспечение системы механизмом мониторинга серверов .....	48
4.1    Проектирование структуры пакетов сетевого протокола .....	48
4.2    Программная реализация механизма мониторинга .....	56
4.3    Программная реализация механизма распределения пользователей по выделенным серверам .....	64
ЗАКЛЮЧЕНИЕ .....	68
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....	69
ПРИЛОЖЕНИЕ А. Графическая часть выпускной квалификационной работы .....	71

## ВВЕДЕНИЕ

Одним из важнейших аспектов работы с UE является запуск выделенных серверов, которые обеспечивают многопользовательскую работу, взаимодействие между клиентами и поддержание сетевой архитектуры. Однако автоматизация процесса запуска и управления такими серверами представляет собой сложную задачу, особенно в контексте масштабируемых и распределенных систем, где необходимо учитывать множество факторов, включая доступность ресурсов, производительность и отклик системы.

На данный момент существует немалое количество программных инструментов, которые позволяют масштабировать любое приложение и автоматизировать запуск процессов, в том числе и для приложения на UE. Ярким примером таких инструментов является Kubernetes, Terraform и Minikube. Однако они могут быть эффективно использованы не для каждой задачи и не для всех типов приложений. Поэтому крайне важно подходить к выбору методов автоматизации запуска приложения и его процессов взвешенно, с учетом специфики задачи; проводить исследовательскую работу в области решения проблемы; итеративно оценивать и внедрять выбранные технологии в проект.

Целью данной работы является разработка системы автоматизированного запуска серверов, которая позволяет инициировать запуск сессий по запросу с клиентского приложения и в автоматизированном режиме распределять нагрузку между запущенными экземплярами приложения. Также система должна обеспечивать прозрачность, предоставляя детализированные логи о каждом процессе и запросе, происходящем в ходе её работы, а также фиксировать эти данные для последующего анализа.

Для достижения данной цели необходимо выполнить следующие задачи: разработать программу менеджера, которая обрабатывает входящие команды от клиентов, запущенных серверов и демона; разработать демона, который обрабатывает входящие команды от менеджера и управляет процессами серверов; разработать сетевой протокол для настройки взаимодействия между

данными сервисами; реализовать механизм авторизации на стороне клиента для контроля доступа к запуску серверов; реализовать механизм распределения подключившихся пользователей между запущенными серверами; обеспечить систему механизмом мониторинга нагрузки через консольный интерфейс.

Авторизация на стороне клиента является важным аспектом, так как она обеспечивает безопасный доступ к системным ресурсам и позволяет гарантировать, что только авторизованные пользователи могут управлять запуском или подключением к выделенным серверам. Авторизация играет критическую роль в предотвращении несанкционированного использования серверных мощностей и защиты пользовательских данных в многопользовательской среде.

Кроме того, внедрение возможностей работы в оффлайн-режиме повышает удобство использования системы для конечных пользователей. Работа в оффлайн-режиме позволяет пользователям сохранять доступ к ключевым функциям и данным, даже в случае временных перебоев с сетью.

## 1 Запуск приложения с помощью технологии контейнеризации

Существует немалое количество программных инструментов, которые используют технологию контейнеризации и позволяют запускать приложение и его процессы в контейнерах. Самым популярным и отлаженным инструментом такого типа является Kubernetes. Kubernetes – это инструмент для управления серверными приложениями, который помогает автоматически запускать, останавливать и распределять нагрузку между серверами.

Для UE Kubernetes может быть полезен при управлении игровыми серверами: он позволяет автоматически запускать новые серверы, когда приходит больше пользователей, и выключать ненужные, когда нагрузка падает. Это экономит ресурсы, упрощает масштабирование и делает систему стабильнее даже при резких скачках нагрузки.

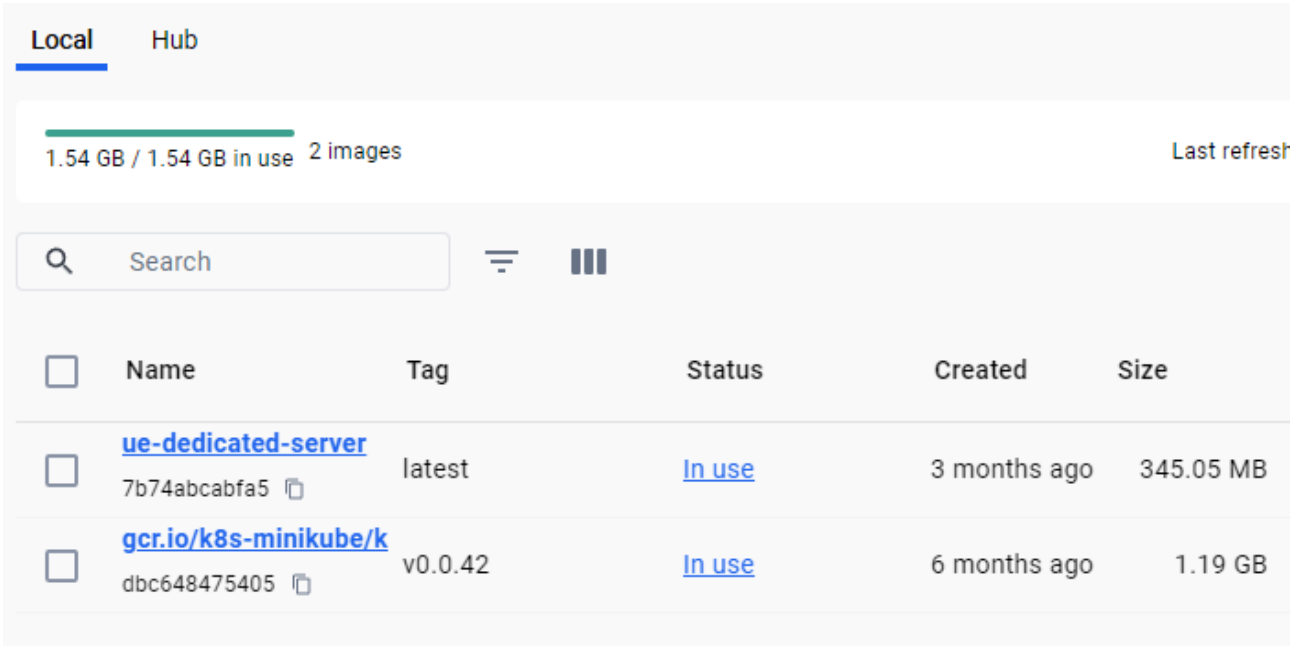
Для того чтобы запустить любое приложение на одном из узлов кластера Kubernetes необходимо для начала контейнеризировать его. Существует два основных способа, которые позволяют запустить сервер в Docker. Первый способ подразумевает использование подготовленного образа Linux-сервера, так как выделенный сервер проекта на UE требует установленного в системе движка, который был собран из исходных кодов. Один из образов, расположенный в системе Docker Hub, который имеет в себе Unreal Engine 4.27, собранный из исходного кода, весит порядка 40 Гб. Второй способ подразумевает сборку выделенного сервера проекта вне контейнера, что позволяет использовать в качестве его образа обычный Linux-сервер.

Первый подход дает возможность автоматизировать сборку выделенного сервера при внесении изменений, однако требует колоссального количества памяти только лишь под один образ контейнера с предустановленным на нем движком UE. Второй подход является более оптимизированным с точки зрения затрачиваемых ресурсов, однако при каждом внесении изменений в проект требует ручной пересборки выделенного сервера.

Для локального тестирования и развертывания приложения с помощью контейнеризации была использована среда Minikube. Minikube – это инструмент,



который позволяет локально запустить Kubernetes на одном узле и управлять контейнерами. Для того чтобы создать образ запускаемого контейнера для начала необходимо собрать выделенный сервер проекта под ядро Linux. Если разработка ведется на операционной системе (ОС) Windows, то для удобства можно воспользоваться официальным инструментом кроссплатформенной сборки под ядро Linux [1]. Общий размер собранного сервера составляет 217 Мб. После сборки сервера необходимо создать Docker-файл. После успешной сборки выделенного сервера под ядро Linux необходимо создать Docker-файл. Docker-файл — это набор команд и инструкций, по которым из образа собирается Docker-контейнер, который в дальнейшем будет размещаться на узле локального кластера Minikube. Вес созданного контейнера из образа выделенного сервера Unreal Engine составляет 345 Мб (рисунок 1).



<input type="checkbox"/>	Name	Tag	Status	Created	Size
<input type="checkbox"/>	<a href="#">ue-dedicated-server</a> 7b74abcbafa5	latest	<a href="#">In use</a>	3 months ago	345.05 MB
<input type="checkbox"/>	<a href="#">gcr.io/k8s-minikube/k</a> dbc648475405	v0.0.42	<a href="#">In use</a>	6 months ago	1.19 GB

Рисунок 1 — Размер созданных образов выделенного сервера и Minikube

На рисунке видно, что размер одного экземпляра выделенного сервера увеличился с 217Мб до 345Мб, что составляет разницу в соотношении размеров в 1,5 раза. С учетом того, что в сетевой инфраструктуре будет использоваться множество контейнеров одновременно, данная разница в размере будет создавать большие накладные расходы. Также образ самого Minikube, который настроен на управление всего лишь одного узла составляет 1,19 Гб. При

использовании кластера Kubernetes в облачной сетевой инфраструктуре данная величина, естественно, будет больше.

Подводя итоги эксперимента с использованием технологии контейнеризации, можно сделать вывод, что Kubernetes, несомненно, является хорошим инструментом для хостинга и масштабирования отказоустойчивого приложения. Использование технологии контейнеризации позволяет избежать такие затратные процессы, как администрирование и настройка Linux-сервера. Однако, его использование не всегда оправдано и имеет ряд минусов.

Во-первых, экземпляр выделенного сервера – это UE приложение, которое выполняет большое количество вычислений для отрисовки графики, симуляции физики и организации сетевого обмена пакетами в режиме реального времени с высокой частотой. Использование контейнеризации создает дополнительный слой абстракции между приложением и аппаратным обеспечением. Это несомненно создает большие накладные расходы при интенсивных вычислениях по сравнению с традиционным запуском приложения без контейнеризации.

Во-вторых, контейнеризация требует значительных вычислительных ресурсов, особенно оперативной памяти. Большее потребление вычислительных ресурсов требует больших затрат на эксплуатацию VPS в облачной инфраструктуре.

Как итог, реализация развертывания и масштабирования с помощью Kubernetes и Docker оправдана в том случае, когда приложение представляет собой высоконагруженный, тяжело масштабируемый сервис с большим сроком жизни, где накладные расходы на создание контейнеров и управление ими играет второстепенную роль в сравнении с основной нагрузкой.

## 2 Разработка собственных сетевых сервисов системы

Главной идеей написания собственных сетевых сервисов является полный отказ от использования средств и инструментов виртуализации таких как Docker и Kubernetes. Вместо этого необходимо будет реализовать такую сетевую инфраструктуру, которая будет в автоматизированном режиме запускать на удаленном сервере или удаленной физической машине сервера UE средствами сетевого программирования и программы-демона.

Явным плюсом такого подхода является полный контроль над узлами в инфраструктуре и значительное уменьшение затрачиваемых вычислительных ресурсов. Прежде всего речь идет об оперативной памяти, так как любые средства виртуализации требуют наличие большего количества оперативной и физической памяти в системе. Ниже представлена рисунок с схемой возможной реализации такой сетевой инфраструктуры (рисунок 2).



Рисунок 2 — Схема сетевой инфраструктуры с использованием собственных сервисов

Анализируя рисунок 2, можно увидеть пример сетевой инфраструктуры, где имеется один узел, на котором расположен менеджер. У каждого клиента налажена двусторонняя связь с менеджером. Каждый клиент представляет собой

приложение на UE, где пользователь может изъявить желание создать новую сессию или присоединиться к уже существующей сессии. Менеджер представляет собой программу, которая обрабатывает запросы клиентов и серверов UE. Таким образом, к основным выполняемым задачам менеджера относятся следующие пункты:

1. Балансировка загрузки. Менеджер, обрабатывая запросы клиентов, должен заносить их в свою внутреннюю структуру данных и обеспечивать подбор игроков. В случае, если в текущий момент времени не создано ни одного экземпляра сервера или все запущенные сервера заняты, менеджер в ответ на входящую команду от клиента должен создать сервер и передать пользователю IP и порт для подключения. В случае, если уже имеются запущенные сервера, менеджер должен найти подходящий по нагрузке экземпляр и передать его IP и порт клиенту для подключения.

2. Отправка команд. Менеджер должен формировать запросы на создание или удаление экземпляров выделенных серверов. Запросы передаются по сети демону, который обрабатывает полученные запросы и удаленно запускает/удаляет экземпляры выделенных серверов приложения на VPS или физической машине.

3. Мониторинг процессов. Менеджер должен отправлять запросы на машину(ы), на которой(ых) запущены процессы выделенных серверов, чтобы выполнять периодическую синхронизацию их состояния (статус работы, активность, количество подключений и многое другое). Далее данная информация обрабатывается и хранится в менеджере.

Двусторонняя связь между клиентом и менеджером устроена следующим образом: когда клиент изъявляет желание создать сессию или присоединиться к существующей сессии, он делает сетевой запрос менеджеру. Тот в свою очередь обрабатывает запрос клиента и заносит его в структуру данных. Менеджер определяет наличие свободных серверов и делает запрос демону на удаленный запуск нового экземпляра выделенного сервера. При успешном запуске выделенного сервера тот посылает обратный запрос менеджеру с информацией

о его IP-адресе и порте, на котором запустился процесс. Менеджер обрабатывает данную информацию и отправляет запрос всем клиентам, которые изъявили желание начать сессию. Клиенты, получив информацию, подключаются по указанному IP-адресу и порту в единое виртуальное пространство.

Существует множество инструментов, которые позволяют реализовать менеджер. В рамках проекта были рассмотрены варианты создания менеджера с помощью UE и сетевой библиотеки Boost.Asio.

## 2.1 Разработка менеджера на Unreal Engine

UE имеет достаточно богатый API для реализации сетевой инфраструктуры. В его арсенале имеется множество заголовочных библиотек, с помощью которых можно настроить прием и отправку пакетов данных по сети. В сетевой инфраструктуре, представленной на рисунке 2 целесообразно использовать транспортный протокол TCP, так как он обеспечивает надежную отправку данных в заданном порядке.

Для обмена данными по сети в менеджер необходимо использовать два вида сокета [2]: сокет для прослушки входящих соединений (LS) и сокет подключений для отправки данных на клиент или демону. Для хранения сокета в UE используется класс *FSocket*.

LS создается с помощью конструктора класса *FTcpSocketBuilder*. Для создания сокета необходимо указать его имя, а также *Endpoint* – IP-адрес и порт, по которому менеджер будет прослушивать входящие соединения. Метод *FTcpSocketBuilder::Listening(int 32 MaxBackLock)* принимает в качестве единственного параметра количество подключений, которое будет вставать в очередь. Если текущее количество подключений превысит заданный порог, они будут отклонены. Далее на основе LS создается новый экземпляр класса *FTcpListener* и оборачивается в класс *TSharedPtr<T>* для лучшей безопасности при доступе.

Важно учесть то, что прослушивание входящих подключений является операцией, которая будет блокировать основной поток, так как в ней содержится исполнение бесконечного цикла. Чтобы избежать блокировку основного потока,

необходимо породить дочерний поток, в котором в “фоновом” режиме будет происходить прослушка входящих соединений.

С помощью статического метода *FRunnableThread::Create(FRunnable\*)* в переменную записывается указатель на созданный поток, где в цикле выполняется логика сокета, записанного в переменную *TcpListener*. Важно понимать, что такое использование приведенного статического метода возможно из-за того, что класс *FTcpListener* наследуется от класса *FRunnable*.

Для того, что создать сокет подключения, необходимо в метод *FTcpListener::OnConnectionAccepted* передать колбэк-функцию вида *OnConnected(FSocket\* ClientSocket, const FIPv4Endpoint& ClientEndpoint)*, в теле которой необходимо записывать сокет подключения в переменную, чтобы получать и отправлять сообщения.

Класс *FReceiveThread* представляет собой класс, унаследованный от класса *FRunnable*. В классе *FRunnable* определены 4 основных метода, которые надо переопределить, чтобы иметь возможность запускать логику работы с сокетом подключения в дочернем потоке [3]. Основной функцией, которую надо переопределить является метод *uint32 Run()*. Как правило, в нее необходимо поместить бесконечный цикл, который будет выполнять прием данных, которые пришли по сокету подключения клиента. Код возврата 0 сигнализирует, что основной цикл успешно выполнил свою работу. При завершении цикла порожденный дочерний поток удаляется. В методе *void Exit()* опционально можно поместить логику по очистке.

Таким образом, создав два вида сокета, менеджер может выполнять базовый функционал, заключающийся в принятии и передачи сообщений по сети с использованием транспортного протокола TCP. Основным минусом реализации менеджера является то, что менеджер реализованный на UE так же потребляет довольно много оперативной памяти, так как движок в режиме реального времени затрачивает вычислительные ресурсы на задачи, не связанные с сетевым обменом данных. В простой базовой версии, где реализовано прослушивание входящих соединений и моментальный ответ на

входящий запрос, менеджер использует 467 Мбайт оперативной памяти (рисунок 3), что довольно много для приложения с таким несложным сетевым функционалом.

Процессы ⊞ + Запустить новую задачу ⊘ Завершить задачу

Имя	Состояние	13% ЦП	28% Память	1% Диск	0% Сеть
> Microsoft Edge (26)		0%	1 437,4 МБ	0 МБ/с	0 Мбит/с
▼ UE4Editor		10,9%	467,7 МБ	0 МБ/с	0 Мбит/с
ServerManager (64-bit De...					
> Telegram Desktop		0%	368,5 МБ	0 МБ/с	0 Мбит/с
> Microsoft Word (32 бита) (6)		0,1%	232,1 МБ	0 МБ/с	0 Мбит/с
> Проводник		0%	178,5 МБ	0 МБ/с	0 Мбит/с
> AMD Ryzen Master		0%	177,6 МБ	0 МБ/с	0 Мбит/с
> Antimalware Service Executable		0%	166,3 МБ	0 МБ/с	0 Мбит/с
Secure System		0%	98,6 МБ	0 МБ/с	0 Мбит/с

Рисунок 3 – Потребление оперативной памяти менеджера, реализованного на UE

Таким образом использование UE для реализации сетевого взаимодействия менеджера серверов с другими узлами сети не оправданно, так как другие сетевые технологии как RPC или репликации, предусмотренные в UE не дают возможности наладить сетевое взаимодействие между двумя разными приложениями UE с разным источником. Также большие затраты оперативной памяти дают понять, что необходимо выбрать альтернативный инструмент для создания менеджера и демона.

## 2.2 Разработка менеджера на C++ и Boost.Asio

Boost.Asio [4] представляет собой заголовочную библиотеку для языка программирования C++, которая позволяет реализовать обмен данными по сети, в том числе по транспортному протоколу TCP. Использование готового решения для реализации сетевого взаимодействия оправдано во избежание возможных ошибок при самостоятельной реализации.

Для начала работы с библиотекой Boost.Asio необходимо скачать исходный код с официального сайта. Как уже было сказано, библиотека является заголовочной, что означает, что работа с некоторыми заголовочными файлами не обязывает разработчика создавать .lib или .dll файлы и настраивать линковщик в настройках проекта. Достаточно лишь добавить дополнительный каталог включаемых файлов в настройки C++ проекта (рисунок 4).

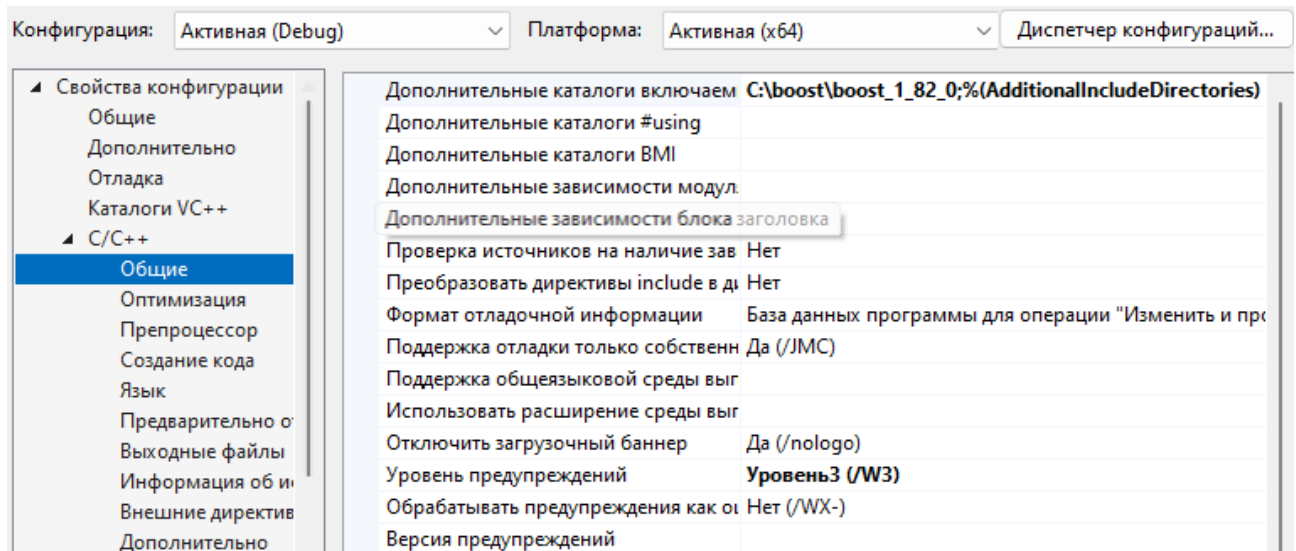


Рисунок 4 — Подключение библиотеки Boost к проекту менеджера

Реализация менеджера на Boost.Asio аналогична реализации на UE. В программе должны быть два вида сокетов [5], один из которых существует на протяжении всей жизни приложения и прослушивает входящие соединения. Второй вид сокета создается при подключении клиента/сервера и позволяет реализовать через себя передачу сообщений по сети. Менеджер будет представлять собой многопоточную синхронную реализацию, так как выполнение синхронных операций более безопасно. Основной цикл программы, или грубо говоря, точка входа программы, представлен ниже (листинг 1).

Листинг 1 — Точка входа программы менеджера

```
void TcpServer::StartServer()
{
    try
    {
        Logger::GetInstance().SetLogFile(_logPath);
    }
    catch (std::runtime_error ex)
    {
        Logger::GetInstance() << std::string(ex.what()) << std::endl;
    }
}
```



```

    Logger::GetInstance() << "Start listening requests on 0.0.0.0:" <<
    _port << std::endl;
    _clientsAcceptThread =
    boost::thread(&TcpServer::CreateClientsAcceptThread, this);
    _serversAcceptThread =
    boost::thread(&TcpServer::CreateServersAcceptThread, this);

    _clientsAcceptThread.join();
    _serversAcceptThread.join();
}

```

В листинге 1 представлено определение метода, где в отдельных дочерних потоках запускается прослушивание входящих сообщений на двух разных сокетах. Первый сокет выполняет прослушивание команд от клиентов, второй – от запущенных серверов. Хотя открытие большого количества портов создает дополнительную уязвимость для несанкционированного доступа, использование двух различных сокетов в данном случае вполне оправдано по нескольким причинам. Во-первых, у клиента и у сервера разный набор команд, который они отправляют менеджеру. Во-вторых, у клиента и у сервера кардинально разная интенсивность в обмене сообщениями. Клиент в случае успешного сценария может отправить команду единожды и закрыть соединение, а сервер в свое время должен постоянно обмениваться сообщениями с менеджером и передавать информацию о своем состоянии и нагрузке.

Прослушивание входящих соединений как от клиента, так и от сервера выполняется в бесконечном цикле. Как и в UE, чтение данных, которые приходят по входящему соединению, также организуется в дочернем потоке, который порождается с помощью статического метода *boost::thread(boost::bind(readDataFromClient, ClientSocket))*. Важно помнить, что выполнение бесконечного цикла, где происходит прослушивание входящих соединений должно быть вынесено в дочерний поток, так как прослушивание подключений по порту не является единственной задачей менеджера и основной поток выполнения программы не должен быть заблокирован.

Как было сказано ранее, для чтения данных из входящего соединения также создается отдельный сокет для принятия сообщений и их чтение происходит параллельно в отдельном потоке, чтобы не блокировать цикл

прослушивания других входящих соединений. Фрагмент метод обработки команд от запущенного сервера представлен ниже (листинг 2).

## Листинг 2 – Метод обработки команд, приходящих от запущенных серверов

```
void TcpServer::ProcessDataFromServer(std::string& message,
boost::shared_ptr<boost::asio::ip::tcp::socket> socket)
{
    ServerCommandType commandType =
CommandsHelper::GetServerCommandType(message);
    std::unordered_map<std::string, std::string> commandKeyValuePairs =
CommandsHelper::GetKeyValuePairs(message);

    switch (commandType)
    {
        case ServerCommandType::REGISTER_SERVER:
            //
            case ServerCommandType::UPDATE_SERVER:
            {
                Logger::GetInstance() << "Updating server job started..." <<
std::endl;
                std::string updatedInstanceUuid =
commandKeyValuePairs["uuid"];

                auto registeredServer = boost::find_if(_runningServers,
[&updatedInstanceUuid](const ServerInfo& serverInfo)
                {
                    return serverInfo.m_uuid == updatedInstanceUuid;
                });
                default:
                    //
            }
    }
}
```

При чтении и выполнении команд важно учитывать следующий факт, который может заметно влиять на производительность. Все данные, включая и тип команды, метод читает и получает из сокета в виде набора байтов и далее конвертирует в строку в формате UTF-8. Важно корректно диссерализовать тип команды, и перевести его, например, в целочисленный тип с перечислением. Сравнение целочисленных типов является гораздо менее затратной операцией, чем сравнение строковых типов. Поэтому, при разработке системы было принято решение проектировать сетевых методов и пакетов с использованием целочисленных типов данных взамен строковых, где это возможно.

На стороне клиента (приложение на UE) необходимо реализовать возможность прием/передачу сообщений по сети также с помощью протокола ТСР. Для отправки сообщений был реализован простой виджет, с управляющим элементом (рисунок 5).

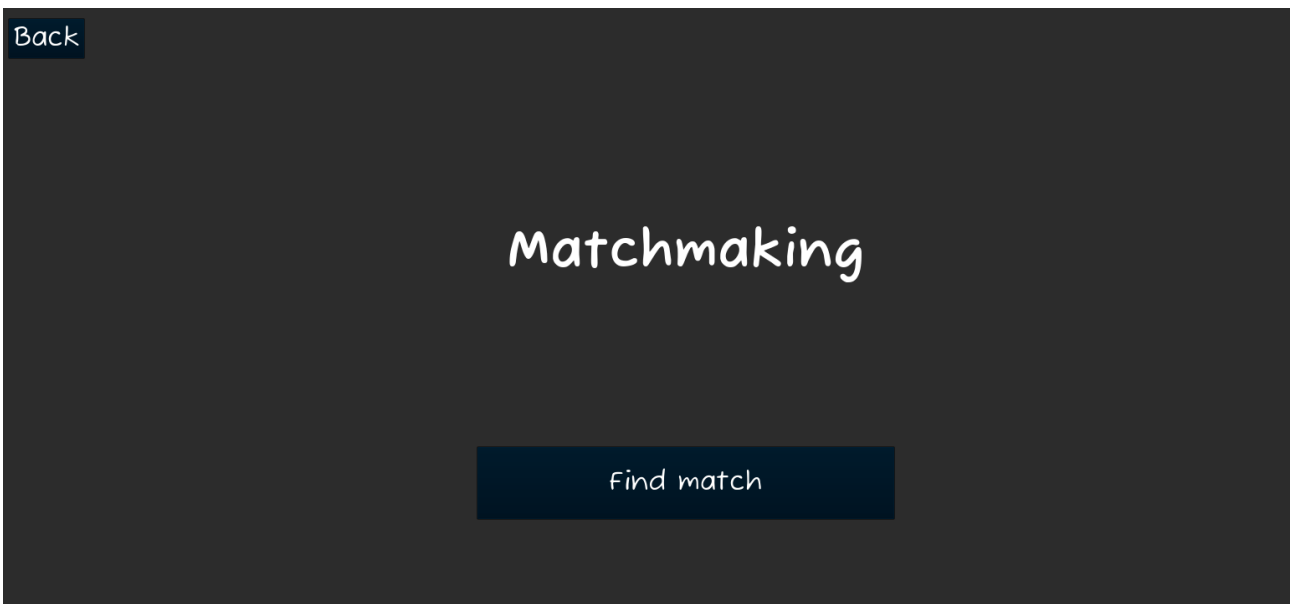


Рисунок 5 — Виджет на клиенте для отправки сообщений в менеджер

По нажатии на элемент виджета “Find match” в менеджер по указанному адресу в конфиге приложения отправляется сообщение в виде обычной строки о том, что пользователь желает создать сессию или подключиться к существующей

Менеджер прослушивает соединения по порту 8870, обрабатывает входящие запросы и выводит отладочное сообщение в консоль.

Таким образом реализованный с помощью сетевой библиотеки для C++ Boost.Asio менеджер серверов взаимодействует с клиентами (приложениями на UE) по сети с использованием транспортного протокола TCP.

### **2.3 Разработка демона на C++ и Boost.Asio**

Демон – это программа, которая запускается в фоновом режиме и выполняет поставленные ей задачи [6]. К основным задачам демона в рамках проекта можно отнести умение принимать команды и реагировать на них; умение запускать по команде экземпляр выделенного сервера UE; умение собирать информацию системы о занятых и свободных вычислительных ресурсах; умение собирать информацию о состоянии всех запущенных выделенных серверов UE.

Для реализации поставленных задач была выбрана библиотека с открытым исходным кодом Boost, которая включает в себя классы и пространства имен для работы с сетевой частью и потоками, необходимые для реализации демона.

Для того, чтобы демон мог принимать входящие сообщения, был создан сетевой сокет, основанный на транспортном протоколе TCP [7]. Прослушивание по сокету является блокирующей операцией из-за наличия бесконечного цикла, поэтому необходимо было создать отдельный поток и инициализировать в нем сокет, аналогично процессу разработки программы менеджера.

С помощью класса *boost::asio::ip::tcp::endpoint* необходимо указать, по какому адресу и порту будет происходить прослушивание входящих соединений. Важно указать именно адрес 0.0.0.0, а не 127.0.0.1, так как при указании адреса 127.0.0.1 сервис будет доступен только в рамках localhost. При указании сетевого интерфейса 0.0.0.0 сервис будет доступен для любого внешнего подключения в пределах локальной сети.

Обработка команд была реализована в методе *handleIncomeQuery*, которая читает сообщения из сокета менеджера, создаваемого для каждого входящего подключения.

При успешном чтении данных из *clientSocket* и получении команды “START”, вызывается метод *StartServerInstance*. Важно отметить, что вызов функции является блокирующей операцией. Для этого, работа функции была вынесена в отдельный поток с помощью экземпляра класса *boost::thread*. Также необходимо было выполнить метод *detach*, который «отсоединяет» указанный поток, что позволяет ему полностью независимо существовать от вызвавшего его основного потока (листинг 3).

### Листинг 3 – Метод запуска процесса выделенного сервера

```
void TcpServer::StartServerInstance()
{
    try
    {
        boost::process::child childThreat(_scriptPath, boost::process::std_out >
std::cout, boost::process::std_err > stderr);
        childThreat.wait();
        if (childThreat.exit_code() == 0)
        {
            Logger::GetInstance() << "Server instance started successfully!" <<
std::endl;
        }
        else
        {
            Logger::GetInstance() << "Error, while starting server instance.
Exit code: " << childThreat.exit_code() << std::endl;
        }
    }
}
```

```

    }
    catch (const std::exception& exception)
    {
        Logger::GetInstance() << "Exception occurred, while starting server
instance: " << exception.what() << std::endl;
    }
}

```

Как показано в листинге выше, запуск экземпляра сервера осуществлялся через создание экземпляра класса *boost::process::child*, который позволяет порождать дочерние процессы. Параметр конструктора *scriptPath* представляет собой путь до скрипта, запускающий процесс выделенного сервера; *boost::process::std\_out > stdout* указывает, что вывод процесса необходимо перенаправлять в стандартный поток вывода; *boost::process::std\_err > stderr* аналогично указывает на необходимость перенаправления потока ошибок.

Процесс запуска для безопасности приложения важно было обернуть в конструкцию *try catch*, так как это потенциально опасное место, где могут возникнуть ошибки и исключения. В блоке *try* демон запускает указанный процесс. При успешном запуске экземпляра выделенного сервера в консоль выводится соответствующее сообщение. Также при возникновении исключений, они обрабатываются в блоке *catch* и выводятся в файл логов для дальнейшей отладки.

Далее, после разработки основного функционала демона необходимо спроектировать и программно реализовать обратное сетевое взаимодействие. Под обратным сетевым взаимодействием подразумевается отправка сообщения с запущенного экземпляра выделенного сервера на клиент UE. Данное сообщение содержит IP-адрес и порт, на котором запущен выделенный сервер, которое необходимо для того, чтобы клиент, который изъявил желание запустить матч, получил необходимую информацию об адресе подключения. В ходе проведения исследовательской работы было выявлено два способа, с помощью которых можно организовать вышеописанное обратное сетевое взаимодействие (рисунок 6)

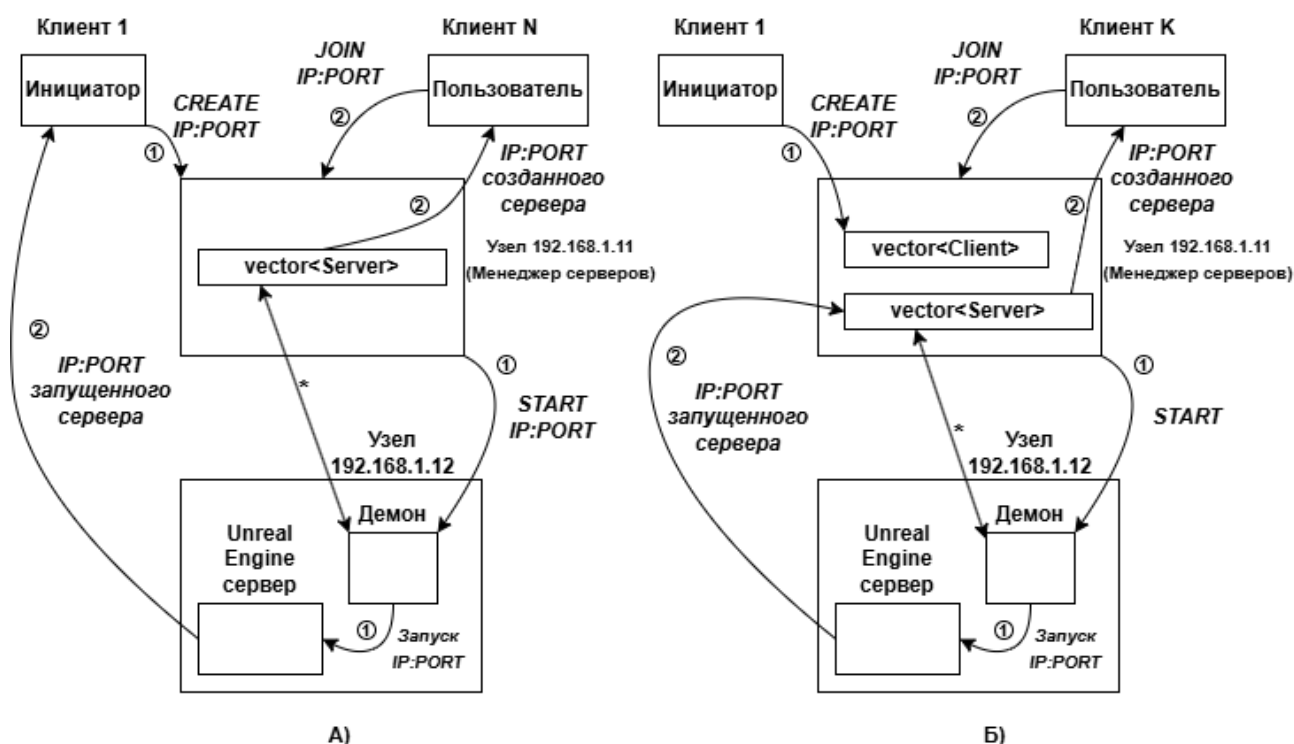


Рисунок 6 – А) Схема обратного сетевого взаимодействия напрямую с клиентом. Б) Схема обратного сетевого взаимодействия через менеджера серверов

Анализируя рисунок 6 можно увидеть два разных подхода к обратной отправке сообщения. Структура данных *vector<Server>* представляет собой список всех работающих экземпляров выделенных серверов. Структура данных *vector<Client>* представляет собой список всех клиентов, которые инициировали соединение с менеджером серверов. В первом подходе IP-адрес и порт запущенного экземпляра выделенного сервера сообщаются клиенту напрямую, без посредников.

К плюсам первого подхода можно отнести большую надежность доставки, так как сообщение проходит меньше сетевых узлов и вероятность потери сетевого пакета уменьшается. К минусам можно отнести отсутствие полного контроля над системой, так как сообщения о запуске сервера и отправке его IP-адреса не логируются централизованно через менеджера серверов. Также к минусам можно отнести вынужденную запись IP-адреса и порта клиента-инициатора в параметры запуска выделенного сервера UE.

Во втором подходе сообщение с IP-адресом и портом передается обратно менеджеру, а не напрямую клиенту, который изъявил желание начать сессию.

К плюсам второго способа можно отнести возможность лучшего контроля над системой, так как все действия с отправкой и записью сообщений централизованно логируются в программе менеджера. К минусам такого подхода можно отнести бóльшее потребление памяти, так как в программе необходимо хранить информацию о клиентах, которые выразили желание начать сессию.

В ходе выполнения исследовательской работы было принято решение в пользу второго способа сетевого взаимодействия, которое обеспечивает более предсказуемое и открытое поведение системы.

Для программной реализации вышеописанного подхода необходимо в классе, производного от *AGameMode*, в методе *BeginPlay* инициализировать отправку сообщения с указанным IP-адресом и портом, на котором запустился сервер. Можно предположить, что IP-адрес узла, на котором запускается выделенный сервер заранее известен в системе и может быть указан в конфиге приложения. Порт, на котором запущено приложение, заранее никогда не известен и должен быть получен программно не из конфига. Получение порта, по которому созданный сервер слушает входящие соединения, было реализовано с помощью метода *LowLevelGetNetworkNumber* класса *UNetDriver*. Данный метод возвращает строку, которую было необходимо разбить по символу разделителю двоеточия. Далее было необходимо получить порт и отправить полученную строку в программу сервера менеджеров. IP-адрес и порт, по которому слушает программа менеджера была записана в конфиг приложения UE.

В программе менеджера необходимо реализовать обработку сообщений с IP-адресом и портом, получаемых от запущенного выделенного сервера. Для этого необходимо определить метод, который получает сообщения по сокету, выбирает нужного клиента из списка сохраненных в памяти приложения и отправляет ему полученное сообщение с информацией о подключении (листинг 4).

## Листинг 4 – Метод обработки информации о подключении от запущенного сервера

```
void TcpServer::SendConnectionStringToClient(std::string& message)
{
    // Обработка присланного URI от DedicatedServer
    Logger::GetInstance() << "Sending uri of started dedicated server to
client: " << message << std::endl;

    auto initiatorConnectedClients =
boost::adaptors::filter(_connectedClients, [](const ClientInfo&
clientInfo)
    {
        return clientInfo.UserType == ClientType::INITIATOR;
    });

    if (initiatorConnectedClients.empty())
    {
        Logger::GetInstance() << "Connected clients queue is empty. No
client to send IP:PORT to" << std::endl;
        return;
    }

    ClientInfo& firstInitiatorInQueue =
initiatorConnectedClients.front();
    Logger::GetInstance() << "Senging data to cleint: " << message <<
std::endl;
    SendDataToSocket(firstInitiatorInQueue.Socket, message);
    _connectedClients.erase(_connectedClients.begin());
}
```

Анализируя листинг, можно заметить, что отправка сообщений клиентам организована в соответствии с принципами работы очереди. Если по команде был запущен выделенный сервер, то необходимо отправить его IP-адрес и порт первому клиенту в очереди, который инициировал данный запуск (клиент с типом *ClientType::INITIATOR*). Ниже представлена UML-диаграмма последовательности для успешного сценария сетевого взаимодействия клиента-инициатора с системой (рисунок 7).



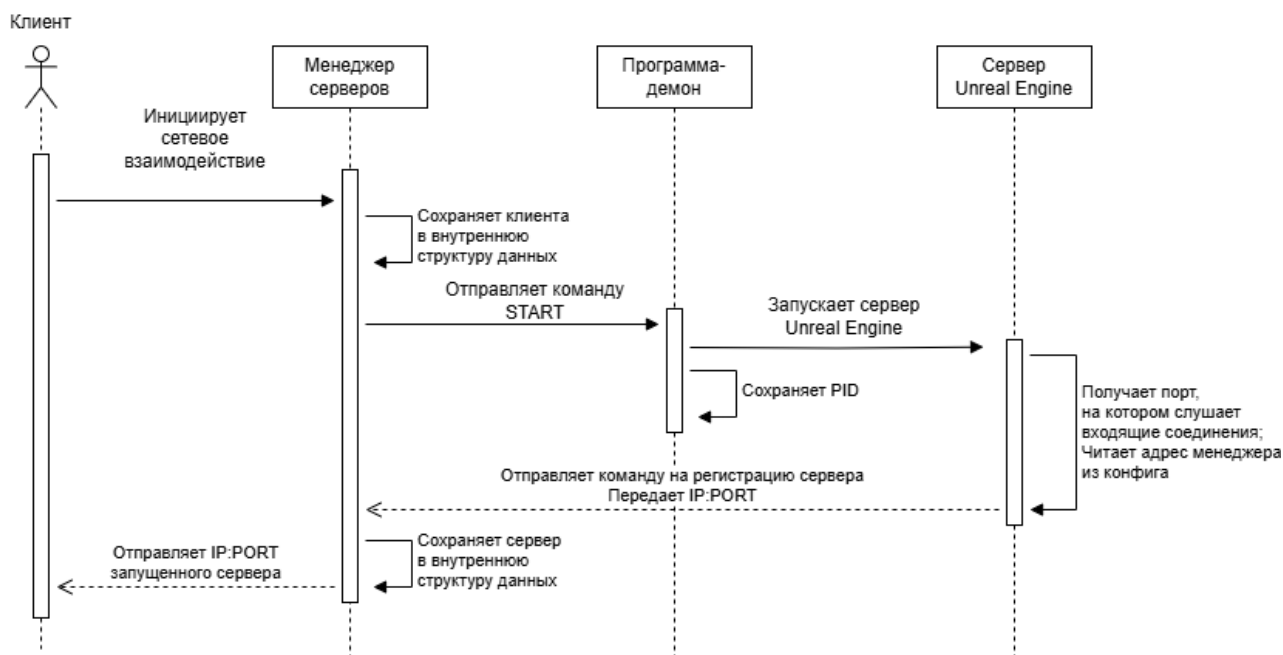


Рисунок 7 – UML-диаграмма последовательности взаимодействия клиента-инициатора с системой (успешный сценарий)

На клиенте была реализована удобная обработка сообщения с информацией о подключении, чтобы пользователь мог подключиться по указанному IP-адресу в автоматическом режиме. Обработка команд была реализована в отдельном потоке, так как прослушивание по сокету является блокирующей операцией. Как только клиент UE получил информацию и обработал ее в дочернем потоке, приложение сразу же подключает пользователя на сервер по указанному IP-адресу и порту. Такой сценарий довольно удобен, так как пользователю ничего не надо нажимать или подтверждать, смена сервера происходит в автоматическом режиме. При этом смена сервера сопровождается информационным сообщением в правом нижнем углу экрана пользователя (рисунок 8). Это действие делает взаимодействие с приложением более нативным и удобным.

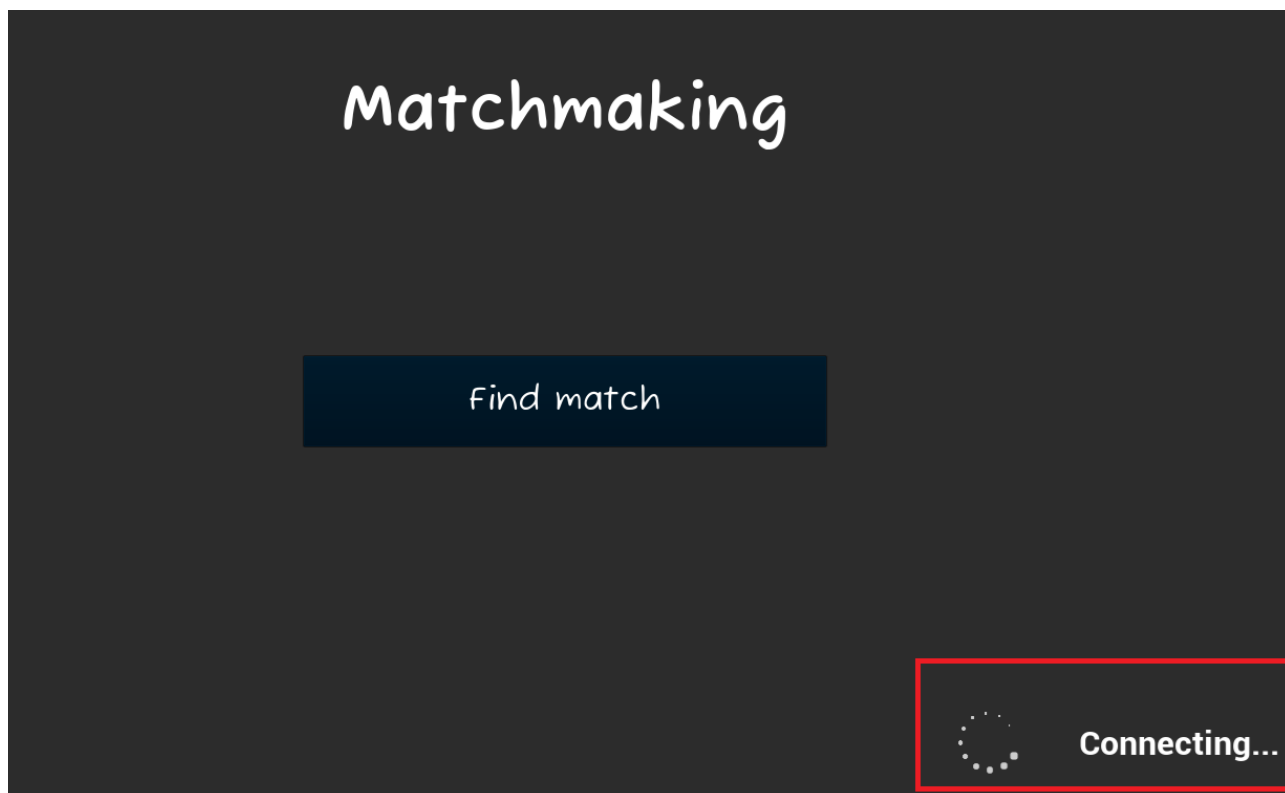


Рисунок 8 – Виджет на экране пользователя при подключении к запущенному серверу

## 2.4 Программная реализация логгера

Для того, чтобы работа системы была понятна разработчику и другим участникам команды, и чтобы работу системы можно было легче отладить, почти каждое происходящее в системе действие, должно быть зафиксировано. Как правило, в качестве места, куда записываются логи выбирают файл, где каждое действие записывается в формате "<TimeStamp>: <Выполненное действие>".

В программах менеджера серверов был реализован простой класс логгера через синглтон Майерса [8]. Синглтон Майерса позволяет через статическую переменную в классе единожды, при первой инициализации, создать экземпляр логгера и переиспользовать уже созданный экземпляр в других местах программы. Статические переменные принадлежат классу, а не одному из его экземпляров. При этом они хранятся не на стеке и не на куче, а в сегменте данных.

Для того, чтобы организовать грамотную и понятную структуру файлов лога, как правило логи записываются не в один большой файл, а разбиваются в соответствии с текущей датой [9]. Таким образом, произошедшие в программе события в один определенный день записываются в файл, каждый файл имеет название текущей даты, например “2025-02-19.log”. Так как каждое действие маркируется временем с точностью до миллисекунд, не составит никакого труда найти лог, соответствующий точной дате и точному времени критической ошибки или существенной неисправности.

Создание файлов логов осуществляется программно и в автоматическом режиме и складывается в директорию Log рядом с исполняемым файлом (листинг 5).

#### Листинг 5 – Метод создания структуры файлов логирования

```
void Logger::SetLogFile(const std::string& logDirectory)
{
    boost::gregorian::date todayDate =
boost::gregorian::day_clock::local_day();
    std::string fileName = boost::gregorian::to_iso_extended_string(todayDate)
+ ".log";

    boost::filesystem::path dirPath =
boost::filesystem::absolute(logDirectory);
    boost::filesystem::path filePath = boost::filesystem::path(logDirectory) /
fileName;

    if (!boost::filesystem::exists(dirPath))
    {
        if (!boost::filesystem::create_directory(dirPath))
        {
            std::string errorMessage = "Failed to create log directory: "
+ dirPath.string();
            throw std::runtime_error(errorMessage);
        }
    }

    fileStream.open(filePath.string(), std::ios::app);
    if (!fileStream.is_open())
    {
        throw std::runtime_error("Failed to open log file");
    }
}
```

На данном этапе при первой инициализации экземпляра логгера и при вызове метода *SetLogFile* указывается путь к директории, куда необходимо сохранять файлы логов. Пример структуры одного из файлов логов представлен ниже (рисунок 9).

```

[2025-03-01 13:00:10.108] Start listening requests on 0.0.0.0:8870
[2025-03-01 13:01:07.146] Got data from client: CREATE
[2025-03-01 13:01:07.148] Sending "START" command to daemon due to empty running server instances array
[2025-03-01 13:01:07.150] Sending command to daemon: START
[2025-03-01 13:01:07.157] Added client to queue: [CLIENT_ADDRESS=127.0.0.1:52252, CLIENT_TYPE=INITIATOR]
[2025-03-01 13:01:08.911] Got data from dedicated server: REGISTER_SERVER,uuid=1D6B4B5D4194BE7B571B349038436F24,uri=127.0.0.1:7777,
[2025-03-01 13:01:08.914] Registering server job started...
[2025-03-01 13:01:08.916] Registered server with uuid = 1D6B4B5D4194BE7B571B349038436F24
[2025-03-01 13:01:08.917] Sending uri of started dedicated server to client: 127.0.0.1:7777
[2025-03-01 13:01:08.919] Sending data to client: 127.0.0.1:7777
[2025-03-01 13:01:08.920] Registering server job finished...
[2025-03-01 13:01:10.843] Got data from dedicated server: UPDATE_SERVER,uuid=1D6B4B5D4194BE7B571B349038436F24,current_players=1
[2025-03-01 13:01:10.845] Updating server job started...
[2025-03-01 13:01:10.847] Old value: 0
[2025-03-01 13:01:10.848] New value: 1
[2025-03-01 13:01:10.850] Updating server job finished...

```

Рисунок 9 – Структура файла логирования программы менеджера

## 2.5 Запуск сервисов на операционной системе Linux Debian

В действительности все самописные сервисы и запущенные сервера UE должны быть запущены на Linux-сервере дистрибутива Debian. Большинство серверов, как правило, запускается именно на дистрибутиве Debian, поскольку данный дистрибутив находится в открытом доступе, достаточно надежен и безопасен, имеет много встроенных инструментов для настройки и администрирования, а также имеет большое сообщество пользователей, системных администраторов и разработчиков.

Важно, что сам дистрибутив будет представлять собой обычный ssh-сервер, который не будет иметь графического интерфейса. Весь процесс взаимодействия с сервером будет происходить исключительно через консольный интерфейс средством выполнения команд.

Далее описан алгоритм запуска программы менеджера. Чтобы запустить приложение на операционной системе Windows под ядро Linux была использована программа виртуализации Virtual Box. Первичная настройка и запуска сервера на виртуальной машине стандартна. Ключевой момент установки заключается в выборе опций программного обеспечения (рисунок 10).

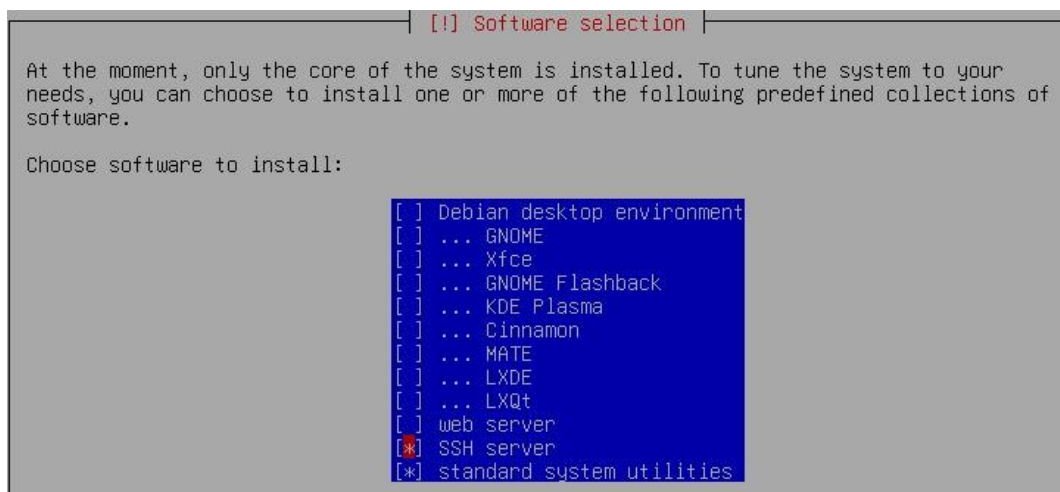


Рисунок 10 – Выбор опций установки гипервизора

Анализируя рисунок 10, можно увидеть, что из всех опций программного обеспечения была выбрана опция установки ssh-сервера и стандартных системных утилит без графического интерфейса. Для того, чтобы виртуальная машина была видна во внутренней сети, необходимо провести первоначальные сетевые настройки гипервизора (рисунок 11).

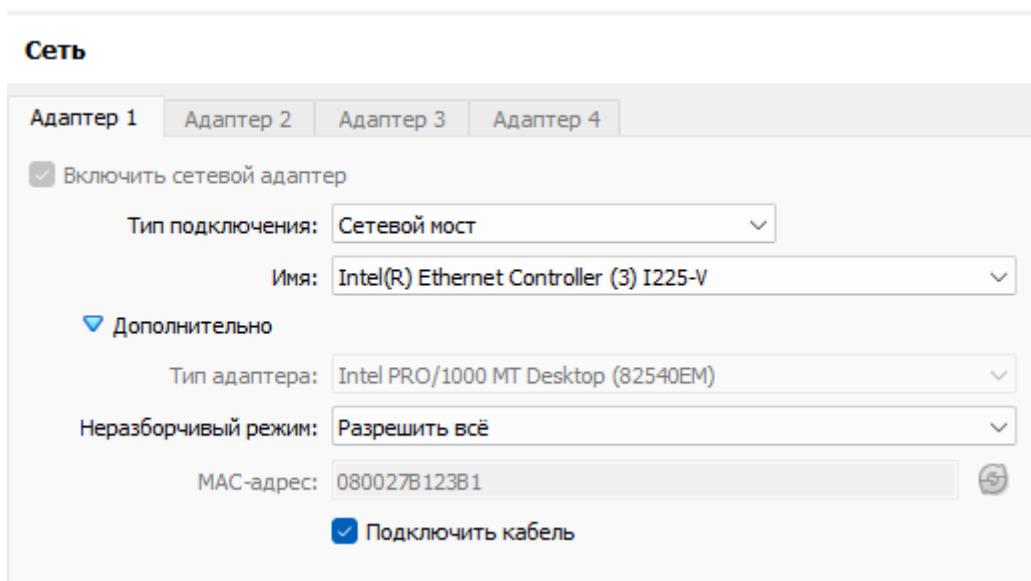


Рисунок 11 – Сетевые настройки гипервизора

Как видно на рисунке 11, в качестве типа подключения необходимо указать “Сетевой мост”; в качестве имени необходимо указать сетевой интерфейс, в данном случае технология *Ethernet*. В пункте “Неразборчивый режим” необходимо указать “Разрешить все”. После запуска ВМ необходимо проверить IP-адрес машины с помощью команды *ip a*. Чтобы понять, что ВМ доступна в

локальной сети и между основной машиной и ВМ доступны сетевые взаимодействия, была выполнена команда *ping 192.168.1.12* с основной машины (рисунок 12).

```
C:\Users\User>ping 192.168.1.12

Обмен пакетами с 192.168.1.12 по 32 байтами данных:
Ответ от 192.168.1.12: число байт=32 время=6мс TTL=64
Ответ от 192.168.1.12: число байт=32 время<1мс TTL=64
Ответ от 192.168.1.12: число байт=32 время<1мс TTL=64
Ответ от 192.168.1.12: число байт=32 время<1мс TTL=64

Статистика Ping для 192.168.1.12:
    Пакетов: отправлено = 4, получено = 4, потеряно = 0
    (0% потерь)
Приблизительное время приема-передачи в мс:
    Минимальное = 0мсек, Максимальное = 6 мсек, Среднее = 1 мсек
```

Рисунок 12 – Выполнение проверки на доступность ВМ в локальной сети

Для того, чтобы запустить программу менеджера на Linux-сервера также необходимо скачать библиотеку *Boost* и провести ее установку [10]. Скачивание стабильной версии библиотеки (архива) было произведено с помощью команды *wget*. Также была выполнена последующая распаковка архива в директорию */home/user/boost/* и выполнение скриптов для установки библиотеки. Для компиляции программы была использована команда

```
g++ -I /home/user/boost/boost_1_82_0 main.cpp Logger.cpp ConfigHelper.cpp
TcpListener.cpp -L /home/boost/boost_1_82_0/stage/lib -lboost_thread -
lboost_system -o server-manager;
```

где *-I* содержит путь к директории с заголовочными файлами библиотеки *Boost*; *-L* содержит путь к директории, который сообщает линковщику путь к скомпилированным файлам *Boost*; *main.cpp* – основной .cpp-файл программы.

При вызове скомпилированного файла программы *./a.out* операционная система может не найти файлы используемой динамической библиотеки. Для этого необходимо скопировать файлы директории, содержащую файлы библиотеки с расширением *.so* в системную директорию */usr/local/lib*. После данных действий операционная система прежде всего будет проверять наличие требуемых файлов динамической линковки в директории */usr/local/lib*. В

результате на Linux-сервере дистрибутива Debian была запущена программа менеджера серверов.

Для компиляции демона необходимо аналогично провести все действия, необходимы для компиляции программы менеджера. Также необходимо повторить установку библиотеки Boost, так как менеджер и демон располагаются на разных узлах. Такое решение было принято вследствие того, что узлу, на котором запускаются сервера UE необходимо наибольшее количество вычислительных ресурсов. Программа менеджера, несомненно, будет потреблять определенную их часть.

Запустить скомпилированную программу, если до этого не указан ключ *-o*, можно запустив исполняемый файл *./a.out &*. Знак *&* означает, что процесс будет выполняться в фоновом режиме и не будет блокировать ввод других команд в терминал. Для того, чтобы убедиться в запуске процесса, можно выполнить команду *ps aux | grep ./a.out*, которая покажет *PID* запущенного процесса. Для завершения процесса использовалась команда *kill -9 <PID>*.

Также, помимо компиляции демона, на узел необходимо установить файлы заранее скомпилированного и собранного выделенного сервера UE. При дальнейшей работе и получении команды “START” демон будет выполнять установленный bash-скрипт и ОС Linux будет запускать процессы выделенных серверов UE.

Было принято решение передать файлы выделенного сервера UE на узел с демоном по протоколу ssh. Для того, чтобы убедиться в работе ssh-сервера на Linux, была использована команда *service ssh status*. Для удаленного копирования с Windows на Linux на основной машине из терминала Power Shell была выполнена команда

```
pscp E:\Master\sem2\MMAPS\Releases\LinuxServer.zip  
user@192.168.1.12:/home/user/dedicated-server
```

Для запуска выделенного сервера UE в целях безопасности нельзя использовать пользователя с правами *root*. Поэтому для этого был создан пользователь *user*, которому были прописаны права *sudo*. Также скрипту,

который запускает экземпляр выделенного сервера, были выданы права с помощью выполнения команды

```
chmod +x /home/user/dedicated-server/LinuxServer/Lab4Server.sh.
```

Финальным шагом развертывания системы на машинах Linux-Debian будет настройка файрволла на обоих узлах. Проблема заключается в том, что для того, чтобы из локальной сети можно было подключиться к процессу демона или менеджера, необходимо изменить правила *firewall* и разрешить прослушивание по указанному порту и протоколу. Для открытия порта была использована программная утилита *firewalld* [11]. Ниже представлен набор команд, который использовался для открытия порта (листинг 6).

Листинг 6 – Список команд для открытия порта сервиса

```
systemctl start firewalld
systemctl enable firewalld
firewall-cmd --permanent --zone=public --add-port=8871/tcp
firewall-cmd -reload
```

Команда *start* запускает сервис, команда *enable* включает сервис при каждой загрузке системы, третья основная команда добавляет правило для порта 8871 и указывает, что прослушивание будет производиться по протоколу TCP. Четвертая команда применяет новые правила и перезапускает сервис. Если добавление было произведено успешно, то команда *firewall-cmd --list-ports* покажет добавленный порт в списке. С помощью команды *netstat -tuln* проверялось, что сервис запущен, слушал входящие соединения и был доступен по сетевому интерфейсу 0.0.0.0 (рисунок 13).

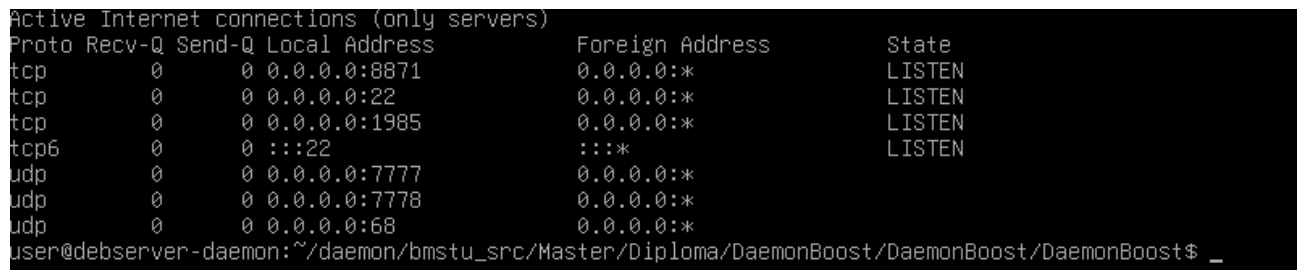
```
user@debserver-daemon:~/daemon/bmstu_src/Master/Diploma/DaemonBoost/DaemonBoost/DaemonBoost$ netstat
-tuln
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:8871             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22               0.0.0.0:*               LISTEN
tcp6       0      0 :::22                   :::*                     LISTEN
udp        0      0 0.0.0.0:68              0.0.0.0:*
user@debserver-daemon:~/daemon/bmstu_src/Master/Diploma/DaemonBoost/DaemonBoost/DaemonBoost$ _
```

Рисунок 13 – Проверка всех сервисов, прослушивающих входящие соединения по протоколам *TCP/UDP*

После выполнения настроек и установок была проверена работоспособность системы. В результаты обработки двух команд “Start”



удаленно были запущены два экземпляра выделенных серверов, как показано на рисунке ниже (рисунок 14).



```
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:8871            0.0.0.0:*              LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*              LISTEN
tcp        0      0 0.0.0.0:1985            0.0.0.0:*              LISTEN
tcp6       0      0 :::22                  :::*                    LISTEN
udp        0      0 0.0.0.0:7777            0.0.0.0:*              LISTEN
udp        0      0 0.0.0.0:7778            0.0.0.0:*              LISTEN
udp        0      0 0.0.0.0:68              0.0.0.0:*              LISTEN
user@debserver-daemon:~/daemon/bmstu_src/Master/Diploma/DaemonBoost/DaemonBoost/DaemonBoost$ _
```

Рисунок 14 – Список запущенных серверов UE на узле Linux Debian

На рисунке 14 видно, что запущено два сервиса на портах 7777/udp (порт по умолчанию для сервера Unreal Engine) и 7778/udp. Также можно заметить, что номер порта увеличивается с количеством запущенных экземпляров. Так как предполагается, что на одной машине с демоном будет запускаться несколько экземпляров серверов UE, для них также были прописаны правила firewall с помощью команды *firewall-cmd --zone=public --permanent --port-add=7777-7797*. Запись команды в таком виде позволяет открыть сразу группу портов 7777-7797, что позволяет запустить до 20 экземпляров UE на одной машине. Также с помощью команды *ps aux* была проверен корректный запуск процессов Unreal Engine. Для проверки доступности запущенных серверов в рамках отладки с клиента UE была выполнена команда *open 192.168.1.12:777* и *192.168.1.12:7778*, которая позволила подключиться к серверам UE, запущенным на VM Linux Debian.

Описанная выше настройка и запуск системы выполнялась вручную. В действительности же, при дальнейшей установке системы на удаленный физический сервер или VPS, такой процесс также необходимо автоматизировать, так как развертывание, установка нужного ПО, компиляция программ занимает довольно долгое время, и почти каждый процесс можно ускорить и автоматизировать.

### **3 Реализация механизма авторизации для контроля запуска серверов**

Авторизация для контроля возможности запуска выделенных серверов в приложении – важный компонент, обеспечивающий доступ к серверным ресурсам только для авторизованных пользователей. Реализация этого механизма может быть выполнена разными способами, включая использование готовых решений, таких как EOS или разработку собственного решения с нуля. Важно учитывать достоинства и недостатки каждого из этих подходов, чтобы выбрать оптимальный метод для конкретного проекта.

Существует два основных подхода для решения задачи авторизации: использование готового API, такого как EOS, или создание собственного решения. EOS предлагает готовый механизм аутентификации и управления покупками с помощью программных интерфейсов `PurchaseInterface` и `StoreInterface`, которые легко интегрируются в любой проект UE и обеспечивают безопасность за счет стандартизированных, протестированных методов. В качестве альтернативы, собственное решение позволяет разработчику самостоятельно настроить всю логику авторизации, учитывая любые специфические требования.

Разработка собственного решения имеет свои плюсы и минусы. К её преимуществам можно отнести полный контроль над процессом аутентификации и возможность гибкой настройки всех аспектов, что особенно полезно при сложных или уникальных требованиях системы. Однако создание собственного решения сопряжено с рисками: велика вероятность появления уязвимостей и багов, которые сложнее выявить и устранить без ресурсов и опыта, как у команды EOS. Самостоятельная разработка потребует больше времени на тестирование и оптимизацию, чтобы соответствовать уровню безопасности, который предлагает EOS.

Подход с использованием EOS также имеет свои особенности. Основное преимущество EOS – это безопасность и стабильность. Платформа прошла множество тестов в реальных условиях, а её функциональность поддерживается

экспертами в области безопасности. Кроме того, EOS обеспечивает быструю и надежную интеграцию, избавляя разработчиков от необходимости проектировать аутентификацию с нуля. Тем не менее, использование EOS может ограничивать гибкость, так как система стандартна и не всегда позволяет детально настроить каждый аспект под уникальные требования.

Таким образом, для задачи аутентификации, контролирующей доступ к выделенным серверам, EOS представляет собой предпочтительное решение. Оно минимизирует риски, связанные с уязвимостями и багами, и предоставляет разработчику надежную, проверенную инфраструктуру.

### 3.1 Программная реализация авторизации

Для начала работы с представленными программными интерфейсами в настройках проекта EOS необходимо добавить предложение. Предложение – это сущность, хранящаяся в системе EOS, которую пользователь может приобрести и получить права на совершение определенного действия. В рамках поставленной задачи наличие у пользователя такого предложения, которое он получил через систему транзакций в интерфейсе проекта, означает, что у него есть право на создание серверов UE. В настройках проекта важно знать ID предложения, которое сгенерировала система EOS, и использовать его для выполнения операций с транзакциями.

Чтобы пользователь мог совершить покупку, он обязательно предварительно должен пройти аутентификацию в системе EOS. В методе *StartPurchase* описан реализованный процесс приобретения предложения и проверки результата транзакции (листинг 7).

Листинг 7 – Реализации транзакции с приобретением предложения

```
void ULab4GameInstance::StartPurchase()
{
    FUniqueNetIdPtr userUniqueId = IdentityPtr->GetUniquePlayerId(0);
    if (!userUniqueId.IsValid()) return;

    if (!UserPurchaseInterface.IsValid()) return;

    FPurchaseCheckoutRequest checkoutRequest = {};
    checkoutRequest.AddPurchaseOffer(TEXT("DedicatedMatchStart"),
    OfferId, 1);
```

```

        UserPurchaseInterface->Checkout(*userUniqueId, checkoutRequest,
FOnPurchaseCheckoutComplete::CreateLambda([this, userUniqueId]
(const FOnlineError& Result, const TSharedRef<FPurchaseReceipt>& Receipt)
{
    if (Result.bSucceeded)
    {
        GetUserReceipts(userUniqueId, true);
        return;
    }

    UE_LOG(LogTemp, Error, TEXT("Failed to complete checkout: %s"),
*(Result.ErrorRaw));
    }));
}

```

Анализируя листинг, можно увидеть, что метод *Checkout* является асинхронным и не блокирует основной поток приложения. Для проверки результата транзакции в параметры функции передается лямбда-функция, которая вызывается при получении результата из системы EOS. В случае успеха транзакции далее выполняется функция *QueryReceipts*, которая устанавливает за пользователем право запускать матч. В случае совпадения ID предложения, найденного в транзакциях, булевая переменная, определенная в классе *UGameInstance*, принимает значение *true*. Важно сохранять такие переменные именно в экземпляр класса, производного от *UGameInstance*, так как экземпляр данного класса существует в памяти на протяжении всего жизненного цикла приложения UE. Управление запуском выделенного сервера осуществляется через виджет, представленный ниже (рисунок 15).



Рисунок 15 – Виджет управления запуском выделенных серверов

Если пользователь авторизовался и ранее не выполнил вышеописанный процесс с транзакцией, в интерфейсе рядом с элементом управления, отвечающим за запуск сервера, будет показано информационное сообщение о необходимости провести процесс аутентификации. При попытке нажать на элемент без подтвержденного права на запуск выделенного сервера система выведет соответствующее предупреждающее сообщение на экран пользователя.

В случае, когда пользователь выполнил процесс приобретения предложения, система удалит подсказку и позволит пользователю запустить выделенный сервер (рисунок 16).

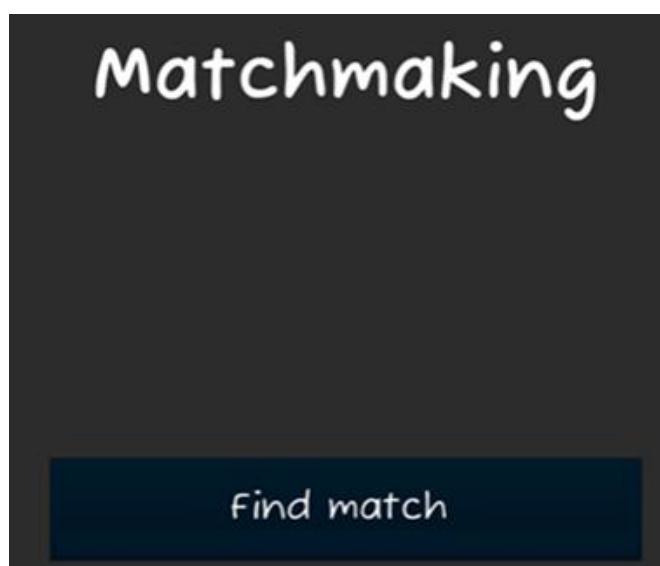


Рисунок 16 – Виджет управления запуском серверов после успешного проведения транзакции

### **3.2 Реализация хранения информации о транзакциях на стороне клиента**

Хранение информации о транзакциях на стороне клиента – это решение, которое часто требует балансировки между безопасностью и удобством для пользователя. С одной стороны, безопасность данных критически важна, особенно если речь идет о сохранении информации о покупках и правах доступа. С другой стороны, удобство пользователей также играет ключевую роль, так как от этого зависит общий опыт работы с приложением. В современных условиях разработки необходимо учитывать оба аспекта, чтобы создать надежное и в то же время дружелюбное пользователю приложение.

Хотя хранение данных о транзакциях на устройстве клиента может показаться менее безопасным решением, этот подход имеет важное преимущество: пользователи могут работать в оффлайн-режиме и сохранять доступ на свои права в приложении без постоянного подключения к сети.

Хранение релевантной информации на стороне клиента сталкивается, как правило, с двумя основными проблемами: защитой от несанкционированного чтения файлов и предотвращением их распространения. Помимо риска того, что третьи лица могут получить доступ к информации о транзакциях, существует и проблема распространения данных — когда сами файлы могут быть перенесены на другое устройство или переданы другим пользователям. Решением для этих задач является использование криптографических методов в сочетании с механизмами, обеспечивающими привязку данных к конкретному устройству. Ниже представлен рисунок с схемой, на которой изображены часто используемые методы защиты от передачи данных между клиентами (рисунок 17). Важно понимать, что это только лишь те методы, которые описываются и открыто разглашаются в сети.

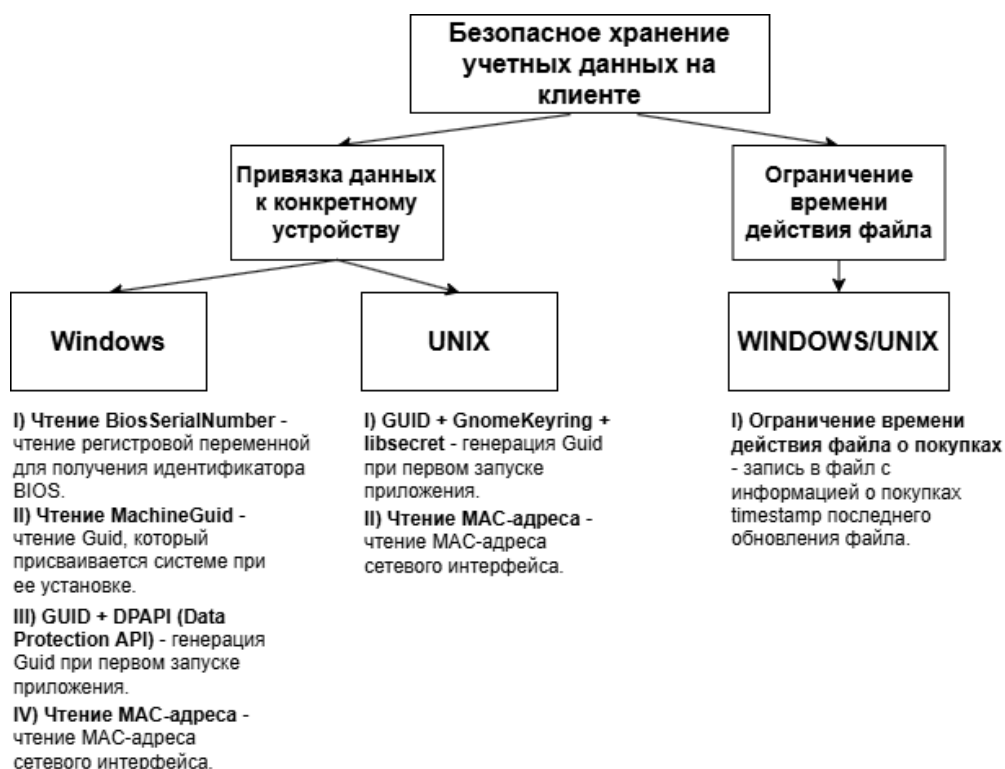


Рисунок 17 – Схема основных способов защиты от распространения файлов между клиентами

На рисунке 17 представлены основные способы защиты от передачи важных файлов между клиентами. Каждый из способов имеет свои преимущества и недостатки. Все способы защиты оценивались с учетом критерия безопасности и неудобств, которые они могут причинить пользователю приложения.

*BIOS Serial Number.* Решение через BIOS Serial Number представляет собой чтение регистровой переменной – уникального идентификатора BIOS. Такой подход обеспечивает доступ к файлам только на исходном устройстве. К преимуществам метода можно отнести то, что он предоставляет надежный уровень защиты, так как файлы будут жестко привязаны только к одному устройству. К недостаткам можно отнести: неудобство для пользователя, если устройство заменено или отремонтировано; ограничение для клиентов на UNIX-подобных системах, так как данный подход будет работать только на системе Windows; требование запуска приложения от имени администратора, так как приложению будет необходимо читать переменные регистра.

*MachineGuid.* Аналогичное решение, как и чтение BIOS Serial Number, которое обладает теми же преимуществами и недостатками.

*MAC-адрес.* Чтение MAC-адреса позволяет однозначно идентифицировать пользователя, так как MAC-адрес уникален для каждого сетевого интерфейса. К преимуществам данного метода можно отнести: работоспособность на устройстве с любой операционной системой. К недостаткам можно отнести: неудобство для пользователя, так как при смене сетевой карты или сетевого интерфейса доступ будет потерян; MAC-адрес гораздо легче подделать, чем BIOS Serial Number или MachineGuid.

*Guid.* Реализация через Guid представляет собой генерацию уникального идентификатора Guid при первом запуске приложения и дальнейшее его хранение в зашифрованном виде в файлах приложениях. Шифрование файлов также можно осуществить несколькими способами. Операционная система Windows предоставляет *Data Protection API*, которое делает возможным шифрование информации на основе данных учетной записи пользователя от

Microsoft. Такой подход к шифрованию обеспечивает надежную защиту, но, очевидно, он не будет работать на клиенте с UNIX-подобной операционной системой. Еще одним очевидным недостатком является то, что при смене учетной записи Microsoft пользователь потеряет доступ к данным файлам, что причиняет неудобство в использовании.

Также ключ шифрования можно хранить в коде самого приложения. Ключ шифрования будет непросто получить несанкционированным способом, если файлы приложения будут доступны пользователю только в виде бинарных файлов без доступа к исходному коду.

*Ограничение времени действия файла.* Подход представляет собой запись в файл информации о последнем обновлении файла, т.е. когда пользователь был последний раз в сети. При каждом обращении к файлу необходимо проверять дату последнего обновления. В случае, если файлы давно не обновлялись, пользователь потеряет к ним доступ. Такой подход играет роль дополнительной защиты от несанкционированного доступа на время. Однако такой подход явно причиняет неудобство пользователю, в случае если он давно не заходил в сеть.

После проведения анализа всех вышеперечисленных методов было принято решение выполнить реализацию именно через генерацию уникального идентификатора Guid, с дальнейшим хранением его в файлах приложения и защитой информации с помощью шифрования. Такой подход не требует жесткой привязки к конкретному устройству, не требует запуска приложения от прав администратора, не привязан к конкретному типу операционной системы. Хотя такой подход и обеспечивает не самую надежную защиту от распространения и прочтения данных несанкционированным способом, однако он обеспечивает соблюдение критерия удобства использования, что, как уже отмечалось выше, является важным аспектом при разработке приложения.

Изначально для реализации описываемого процесса по использованию информации о транзакциях в оффлайн-режиме необходимо сгенерировать уникальный идентификатор приложения – Guid. Вероятность того, что в мире будет сгенерировано два одинаковых ключа крайне мала. Поэтому Guid является



отличным решением в качестве использования уникального идентификатора для приложения.

Если приложению при инициализации не удалось успешно найти файл по пути */Saved/AppData/AppGuid.dat*, то считается, что Guid еще не был сформирован и записан в файл (листинг 8).

#### Листинг 8 – Генерация уникального идентификатора приложения

```
void ULab4GameInstance::GenerateGuidAndSave()
{
    FString appGuidString = FGuid::NewGuid().ToString();
    UE_LOG(LogTemp, Log, TEXT("Generated app GUID"))

    SaveBase64EncodedData(appGuidString,
FPaths::Combine(FPaths::ProjectSavedDir(), TEXT("AppData"),
TEXT("AppGUID.dat")));
}

void ULab4GameInstance::SaveBase64EncodedData(const FString& Data, const
FString& FilePath)
{
    FString encodedData = FBase64::Encode(Data);
    FString filePath = FilePath;

    FFileHelper::SaveStringToFile(encodedData, *filePath);
    UE_LOG(LogTemp, Log, TEXT("Saved encoded GUID to directory"))
}
```

Также стоит отметить, что директория *Saved* была выбрана для хранения не случайным образом. Данная директория обычно содержит все не бинарные файлы, которые UE самостоятельно сохраняет и использует при работе. Данная директория никогда автоматически не очищается движком, следовательно, исключается шанс, что релевантная информация о покупках может быть удалена программно.

Запись информации о транзакциях выполняется сразу после того, как пользователь совершил транзакцию, и при первом запуске приложения, когда пользователь вошел в режим онлайн. Для каждой транзакции генерируется отдельный файл (листинг 9).

## Листинг 9 – Сохранение информации о транзакции в отдельный файл

```
void ULab4GameInstance::SavePurchaseToFile(const FString&
PurchaseOfferId)
{
    if (!CheckIfGuidExists())return;

    FString playerLogin = GetPlayerName();
    FString offerIdHash = GetSHA256Hash(PurchaseOfferId);
    FString appGuid =
LoadBase64EncodedData(FPaths::Combine(FPaths::ProjectSavedDir(),
TEXT("AppData"), TEXT("AppGUID.dat")));

    FString fileName = offerIdHash + TEXT("_") + playerLogin +
TEXT(".dat");

    FString filePath = FPaths::Combine(FPaths::ProjectSavedDir(),
TEXT("Purchases"), fileName);

    if(FPlatformFileManager::Get().GetPlatformFile().FileExists(*filePa
th))
    {
        UE_LOG(LogTemp, Log, TEXT("Offer with ID: %s already exists,
hash: %s. Cancel saving data..."), *PurchaseOfferId, *offerIdHash);
        return;
    }

    FString fileContent =
FString::Printf(TEXT("AppGUID=%s\nOfferId=%s"), *appGuid,
*PurchaseOfferId);

    SaveBase64EncodedData(fileContent, filePath);
}
```

В содержимое файла записывается Guid приложения, которое было сгенерировано ранее и ID предложения, которое было приобретено в рамках транзакции. Для того, что решить проблему уникальности файлов, было принято решение записывать имя файла в формате "<OfferID>\_<PlayerID>". Причем OfferID следует хранить не в открытом виде в названии файла, а, например, в виде контрольной суммы SHA256.

Когда пользователь не прошел аутентификацию в системе EOS и нажимает на кнопку создать матч в оффлайн-режиме, система проверяет наличие файлов с информацией о транзакции с нужным OfferID в директории */Saved/Purchases/*.

Для дополнительной защиты пользователь должен ввести свой логин, который соответствует его логину в системе EOS. Введенный пользователем логин сравнивается с логином, который предоставила система EOS на момент создания и записи файла о транзакции в онлайн режиме. Хотя это отнюдь не

полноценная аутентификация, но она может служить дополнительным препятствием для тех пользователей, кто пытается подделать учетную запись или обойти привязку данных к пользователю. Это может быть полезно в том случае, если пользователь часто предпочитает решения, которые не требуют лишних усилий. Добавление логина в имя файла и проверка его при входе в оффлайн-режиме может использовать этот принцип в качестве дополнительной меры защиты. Большинство пользователей будут интуитивно использовать свой обычный логин, не задумываясь о его точном написании или возможных вариациях, что осложняет попытки доступа к чужим данным. Такая проверка добавляет уровень "ленивой безопасности", поскольку пользователям проще ввести свой привычный логин, чем пытаться обойти систему.

## **4 Обеспечение системы механизмом мониторинга серверов**

Для эффективной балансировки количества игроков по серверам и распределения серверов по виртуальным машинам необходимо создать фундаментальную инфраструктуру мониторинга и управления. Такой подход требует реализации системы, которая будет отслеживать состояние серверов и виртуальных машин, а также обеспечивать их регистрацию в менеджере серверов. Эти шаги являются ключевыми для сбора данных о нагрузке, доступных ресурсах и состоянии элементов инфраструктуры, что позволит в дальнейшем принимать оптимальные решения для балансировки и масштабирования.

### **4.1 Проектирование структуры пакетов сетевого протокола**

Для начала реализации такого требования было принято решение разделить обработку клиентских сокетов и сокетов серверов UE. Это обусловлено тем, что данные типы сокетов имеют различную логику обработки: клиентские сокет и серверные сокет обмениваются данными с разной интенсивностью и требуют специфических подходов к обработке сообщений. Такое разделение позволяет оптимизировать работу с каждым типом соединения и эффективно управлять нагрузкой.

Обмен данными с сервером и менеджером осуществляется на основе долгоживущего TCP сокета (long-lived-socket). Его характерной чертой является то, что он сохраняет соединение на протяжении всей сессии, пока оно явно не разрывается одной из сторон (клиентом либо сервером). Этот подход позволяет избежать создания нового сокета для каждого сообщения, обеспечивая более эффективное взаимодействие и снижая накладные расходы на установление соединения. Для начала передачи сообщений по протоколу TCP клиентская сторона устанавливает соединение с принимающей стороной. Как только соединение было установлено сервер записывает в память сокет клиента и в бесконечном цикле пытается прочитать сообщения из клиентского сокета. Пока попытки чтения не прекращаются, клиентский сокет никогда не будет явно пытаться разорвать существующее соединение. Как только клиент присылает

сообщение, сервер обрабатывает его в отдельном потоке и далее продолжает пытаться слушать следующие входящие сообщения. Если клиентский сокет явно разорвал соединение с сервером, то вызовется исключение типа *boost::system::system\_error*, управление передается в блок *catch* и сервер явно разрывает соединение для двоих сторон выполнив функции *socket->shutdown(boost::asio::ip::tcp::socket::shutdown\_both)* и *socket->close()*.

Для обработки команд, которые приходят от запущенного выделенного сервера, необходимо решить, каким образом проводить десериализацию данных. В задачах передачи данных по сети между компонентами распределенной системы важно обеспечить корректную сериализацию и десериализацию пакетов. Существуют разные подходы реализации данной задачи. Например, (де)сериализация может производиться с использованием текстовых данных в стандартизированных форматах передачи данных как JSON или XML. Или же может быть разработан собственный протокол следующего вида: *<имя\_команды>, <ключ1=значение1>, ..., <ключN=значениеN>*. Первым словом до разделителя, всегда идет тип команды, которую надо обработать. Далее через запятую идут пары ключ значения, которые являются опциональными и предназначены для передачи дополнительной информации к выполняемой команде. Пример команды, которая отправляется с выделенного сервера UE после его запуска для его регистрации в системе имеет вид

*REGISTER\_SERVER,uuid=ae34b65e4a45cd1a2,uri=127.0.0.1:7777,current\_players=1,max\_players=10*

В команде приходят такие данные как *uuid* – уникальный идентификатор сервера, *uri* – адрес подключения, *current\_players* – текущее количество игроков, *max\_players* – максимально возможное количество игроков.

К преимуществам таких текстовых подходов передачи данных можно отнести удобство отладки и мониторинга, хорошую читаемость данных и простоту решения. К главным же недостатком такого подхода можно отнести значительное замедление производительности из-за затрат на парсинг строковых команд; увеличенный размер передаваемых данных и большая вероятность

возникновения ошибки при заполнении сообщения протокола вручную.

Альтернативным подходом является побайтовая (де)сериализация. Преимущества бинарной (де)сериализации включают в себя высокую скорость (де)сериализации, малый размер передаваемых пакетов за счет ручного управления размерами каждого поля и строгую структурированность данных. К недостаткам бинарной сериализации и десериализации, несомненно, относятся трудности при отладке и проблемы совместимости между разными архитектурами из-за разной размерности типов данных и разного порядка байтов. Однако описанную проблему необходимо и возможно разрешить. Следовательно, для принятия решения о используемом подходе передачи данных необходимо было провести исследование, провести анализ возможного использования каждого из подходов и выполнить замеры производительности.

Для строковой сериализации данных достаточно лишь сформировать строку, перекодировать полученную строку в массив байтов и отправить полученную полезную нагрузку по сокету (листинг 10). Клиент, в данном случае выделенный сервер UE, не должен беспокоиться о порядке передачи байтов, так как все заложено в строке.

#### Листинг 10 – Метод отправки строковых данных в менеджер

```
void AEmptyLobbyGameMode::SendMessageWithSocket(const FString& Message)
{
    TArray<uint8> payload;
    FStringToByteArray(Message, payload);
    FBufferArchive ArchiveBuffer;
    int32 bytesSent = 0;
    ArchiveBuffer.Append(payload);
    bool sendResult = ConnectionSocket->Send(ArchiveBuffer.GetData(),
    ArchiveBuffer.Num(), bytesSent);

    if (bytesSent != ArchiveBuffer.Num() || !sendResult)
    {
        UE_LOG(LogTemp, Error, TEXT("Error while submitting message to
    ServerManager)
        return;
    }
}
```

Для сериализации данных на стороне менеджера (листинг 11) достаточно просто записать полученные из сокета байты в строку и далее производить ее парсинг, например, по символу разделителю '='.

## Листинг 11 – Десериализация строковых данных на стороне менеджера

```
while (true)
{
    char data[512];

    size_t bytesRead = socket->read_some(boost::asio::buffer(data));
    if (bytesRead > 0)
    {
        std::string message = std::string(data, bytesRead);
        ProcessDataFromServer(message, socket);
    }
}
```

Также помимо регистрации сервера необходимо было обработать событие, когда меняется текущее количество игроков. В классе *AGameMode* есть виртуальный метод *virtual void PostLogin(APlayerController\* NewPlayer)*, который вызывается каждый раз как к серверу подключается новый пользователь. Аналогичную логику необходимо было реализовать и тогда, когда пользователь отключается от сервера. Для реализации данной задачи был переопределен и использован метод *virtual void Logout(AController\* Exiting)*, который выполняется каждый раз, как пользователь отключается от сервера. Пример логов менеджера, когда сервер был запущен и к нему подключилось два пользователя представлен ниже (рисунок 18).

```
3 [2025-03-18 23:55:55.893] Starting applicaton...
4 [2025-03-18 23:55:55.903] Start listening requests on 0.0.0.0:8870
5 [2025-03-18 23:57:03.406] Got data from client: CREATE
6 [2025-03-18 23:57:03.410] Sending "START" command to daemon due to empty running server instances array
7 [2025-03-18 23:57:03.412] Sending command to daemon: START
8 [2025-03-18 23:57:03.418] Added client to queue: [CLIENT_ADDRESS=127.0.0.1:55591, CLIENT_TYPE=INITIATOR]
9 [2025-03-18 23:57:14.754] Getting data from dedicated server: REGISTER_SERVER,uuid=1D6B4B5D4194BE7B571B349038436F24
10 [2025-03-18 23:57:14.759] Registering server job started...
11 [2025-03-18 23:57:14.766] Registered server with uuid = 1D6B4B5D4194BE7B571B349038436F24
12 [2025-03-18 23:57:14.767] Sending uri of started dedicated server to client: 127.0.0.1:7777
13 [2025-03-18 23:57:14.768] Sending data to cleint: 127.0.0.1:7777
14 [2025-03-18 23:57:14.770] Registering server job finished...
15 [2025-03-18 23:57:15.376] Getting data from dedicated server: UPDATE_SERVER,uuid=1D6B4B5D4194BE7B571B349038436F24
16 [2025-03-18 23:57:15.379] Updating server job started...
17 [2025-03-18 23:57:15.381] Old value: [current_players = 0, state = LOBBY]
18 [2025-03-18 23:57:15.383] New value: [current_players = 1, state = LOBBY]
19 [2025-03-18 23:57:15.384] Updating server job finished...
20 [2025-03-18 23:57:45.270] Got data from client: CREATE
21 [2025-03-18 23:57:45.272] Found server instance [uuid=1D6B4B5D4194BE7B571B349038436F24] for client [CLIENT_ADDRESS=127.0.0.1:55624, CLIENT_TYPE=INITIATOR]
22 [2025-03-18 23:57:45.274] Sending data: [URI=127.0.0.1:7777] to cleint: [CLIENT_ADDRESS=127.0.0.1:55624, CLIENT_TYPE=INITIATOR]
23 [2025-03-18 23:57:46.084] Getting data from dedicated server: UPDATE_SERVER,uuid=1D6B4B5D4194BE7B571B349038436F24
```

Рисунок 18 – Лог менеджера при подключении пользователей  
(строковая десериализация)

На рисунке видно, что при одном из запусков при строковой десериализации с учетом ввода вывода данных в поток процесс регистрации сервера занимает 12 мс и процесс обновления информации о сервере занимает

8 мс. Данное значение невелико, однако, стоит его сравнить с байтовой реализацией.

Для реализации байтовой (де)сериализации для начала необходимо определить тип каждого поля и его размер. Важно отметить, что структура данных и размер каждого поля должны быть одинаковыми на принимающей и отправляющей стороне. Без выполнения этого условия невозможно будет гарантировать грамотную десериализацию данных. Пример структуры данных, который необходимо (де)сериализовать и передавать по сети представлен ниже (листинг 12).

#### Листинг 12 – Структура данных команды регистрации сервера

```
#pragma pack(push, 1)
struct ServerRegisterMessage
{
    uint32_t m_ip;
    uint16_t m_port;
    uint8_t m_uuid[16];
    uint16_t m_currentPlayers;
    uint16_t m_maxPlayers;
    uint8_t m_serverState;
};
#pragma pack(pop)
```

На листинге видно, что используется такие типы данных как *uint16\_t*, *uint8\_t*. Данные типы данных находятся в заголовочном файле библиотеки *cstdint*, и они предназначены для чего, чтобы четко указать размер поля в структуре. К примеру, поле *m\_serverState* – это небольшой *enum*, который содержит в себе перечисление возможных состояний запущенного сервера UE. Данное количество строго ограничено, поэтому для его хранения будет достаточно одного байта. Количество подключенных пользователей – это не строго ограниченное по размерам поле, оно может принимать разные значения. Следовательно, размер поля в два байта обеспечивает достаточный диапазон значений (0 - 65536) для правильного хранения. Поле *m\_uuid* представляет собой UUID сервера, который имеет всегда одинаковую длину в 36 символов. Такой идентификатор можно хранить в виде строки, что будет занимать 37 символов с учетом терминального символа строки '\0'. Гораздо эффективней будет хранить



данное поле в виде 16 байт, так как для представления одного числа в шестнадцатеричной системе счисления необходимо 4 бита. Также можно поступить и с передачей адреса и порта. Гораздо эффективней хранить IP-адрес в виде 4 байт, а порт – в размере 2 байт, так как максимального значения в 65 536 будет вполне хватать для указания порта, на котором запустился сервер. *#pragma pack (push, 1)* и *#pragma pack (pop)* являются директивой препроцессора и служат для того, чтобы указать компилятору на необходимость отключения байтового выравнивания на данном участке кода.

Выравнивание – это правило, по которому компилятор размещает данные в памяти не вплотную, как этого ожидает программист, указав нужный ему размер переменной, а с определенными промежутками. Это позволяет процессору обеспечить чуть более эффективную работу. Например, если структура состоит из двух полей *char a* и *int b*, то структура будет занимать в памяти не 5 байт, как ожидает того программист, а 8, так как компилятор автоматически вставил между данными полями 3 пустых байта для выравнивания. Такой подход делает возможной десериализацию байтовых данных на процессорах с разной архитектурой.

Для сериализации данных на отправляющей стороне достаточно выделить память под нужную структуру данных, заполнить ее полезной нагрузкой и записать в массив байт, который потом передается по сокету (листинг 13).

### Листинг 13 – Байтовая сериализация на стороне сервера UE

```
FIPv4Address ipAddr;
FIPv4Address::Parse(*DaemonAddress, ipAddr);

FGuid appGuid = GetServerInstanceUuid();
FMemory::Memcpy(&(payload.Uuid), &appGuid, 16);

payload.Ip = ipAddr.Value;
payload.Port = Port;
payload.CurrentPlayers = GetNumPlayers();
payload.MaxPlayers = 10;
payload.ServerState = static_cast<uint8_t>(ServerState::LOBBY);

MessageFrameHeader frameHeader;

frameHeader.CommandType =
static_cast<uint8_t>(ServerCommandType::REGISTER_SERVER);
frameHeader.PayloadSize = sizeof(payload);
```

```
TArray<uint8> buffer;
buffer.Append((uint8*)&frameHeader, sizeof(frameHeader));
buffer.Append((uint8*)&payload, sizeof(payload));
```

В листинге выше видно, что помимо полезной нагрузки в сокет также отправляется структура данных *MessageFrameHeader* (метаинформация). Данная структура состоит из поля *uint8\_t CommandType*, обозначающим тип отправляемой команды весом в один байт; и поля *uint16\_t PayloadSize*, обозначающим размер полезной нагрузки в байтах, которая идет следом за метаинформацией. В итоге ожидаемый размер структуры для команды регистрации представлен ниже (рисунок 19).

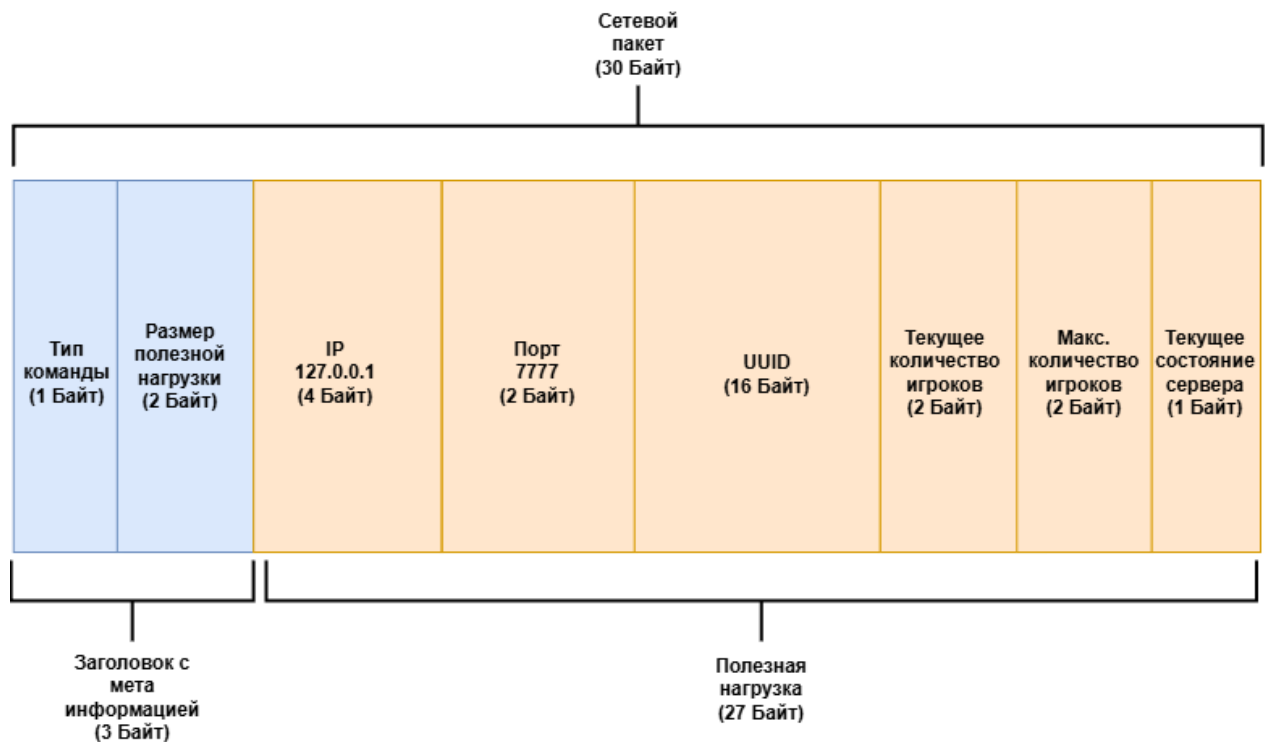


Рисунок 19 – Побайтовая схема сообщения регистрации сервера

С использованием такой структуры байтов в сетевом пакете при получении данных из сокета принимающая сторона понимает, какая команда пришла, какой размер полезной нагрузки, сколько памяти надо выделить под полезную нагрузку, что в дальнейшем ее десериализовать (листинг 14).

Листинг 14 – Десериализация данных на стороне менеджера

```
try
{
    while (true)
    {
```

```

        MessageFrameHeader frameHeader;
        boost::asio::read(*socket, boost::asio::buffer(&frameHeader,
sizeof(MessageFrameHeader)));

        std::vector<char> payload(frameHeader.m_payloadSize);
        boost::asio::read(*socket, boost::asio::buffer(payload.data(),
payload.size()));

        ProcessBinaryDataFromServer(frameHeader, payload);
    }
}

```

При десериализации полезной нагрузки и разбиении ее на поля необходимо обязательно проверить, совпадает ли размер полученной нагрузки с ожидаемой. Если размер совпадает, то можно предпринять попытку копирования массива байтов в структуру данных, иначе – необходимо пропустить полученную команду и вывести соответствующее сообщение (листинг 15).

Листинг 15 – Обработка массива байт полезной нагрузки на стороне менеджера

```

if (payload.size() < sizeof(ServerRegisterMessage))
{
    Logger::GetInstance() << "Invalid payload size for REGISTER_SERVER
command. Finishing register server job" << std::endl;
    return;
}

ServerRegisterMessage newServerRaw;
memcpy(&newServerRaw, payload.data(), sizeof(ServerRegisterMessage));
ServerInfo newServer = ServerInfo::FromRaw(newServerRaw);

```

Ниже на рисунке представлен лог менеджера при реализации байтовой (де)сериализации (рисунок 20).

```

[2025-03-27 14:56:37.471] Starting applicaton...
[2025-03-27 14:56:37.481] Start listening requests on 0.0.0.0:8870
[2025-03-27 14:57:06.849] Got data from client: CREATE
[2025-03-27 14:57:06.851] Sending "START" command to daemon due to empty running server instances array
[2025-03-27 14:57:06.853] Sending command to daemon: START
[2025-03-27 14:57:06.859] Added client to queue: [CLIENT_ADDRESS=127.0.0.1:53912, CLIENT_TYPE=INITIATOR]
[2025-03-27 14:57:08.378] Start processing data from dedicated server
[2025-03-27 14:57:08.379] Registering server job started...
[2025-03-27 14:57:08.380] Registered server with uuid = 1D6B4B5D4194BE7B571B349038436F24
[2025-03-27 14:57:08.381] Sending uri of started dedicated server to client: 127.0.0.1:7777
[2025-03-27 14:57:08.383] Senging data to cleint: 127.0.0.1:7777
[2025-03-27 14:57:08.384] Registering server job finished...
[2025-03-27 14:57:09.680] Start processing data from dedicated server
[2025-03-27 14:57:09.682] Updating server job started...
[2025-03-27 14:57:09.683] Old value: [current_players = 0, state = LOBBY]
[2025-03-27 14:57:09.684] New value: [current_players = 1, state = LOBBY]
[2025-03-27 14:57:09.685] Updating server job finished...
[2025-03-27 14:57:38.921] Got data from client: CREATE
[2025-03-27 14:57:38.924] Found server instance [uuid=1D6B4B5D4194BE7B571B349038436F24] for client [CLIENT_ADDRESS=127.0.0.1:53941, CLIENT_TYPE=PLAYER]
[2025-03-27 14:57:38.926] Senging data: [URI=127.0.0.1:7777] to cleint: [CLIENT_ADDRESS=127.0.0.1:53941, CLIENT_TYPE=PLAYER]
[2025-03-27 14:57:39.635] Start processing data from dedicated server
[2025-03-27 14:57:39.637] Updating server job started...
[2025-03-27 14:57:39.639] Old value: [current_players = 1, state = LOBBY]
[2025-03-27 14:57:39.640] New value: [current_players = 2, state = LOBBY]
[2025-03-27 14:57:39.641] Updating server job finished...

```

Рисунок 20 – Лог менеджера при подключении пользователей  
(байтовая десериализация)

На рисунке видно, что процесс регистрации сервера занимает 1-2 мс с учетом вывода в поток; процесс обновления информации о сервере занимает 3-4 мс.

Таким образом, в результате исследования удалось реализовать байтовый тип (де)сериализации, который значительно быстрее и производительнее, чем строковый тип (де)сериализации. Окончательный выбор реализации обработки данных был выбран в пользу байтовой (де)сериализации, так это позволило получить значительный прирост в скорости обработки команд, что критически важно для низкоуровневых приложений такого типа.

## **4.2 Программная реализация механизма мониторинга**

Современные приложения могут отображать информацию пользователю или администратору разными способами: от консольных интерфейсов CLI до графических интерфейсов GUI. Графические интерфейсы обладают несомненными достоинствами — визуальной наглядностью, удобством взаимодействия и широкими возможностями отображения сложных данных. Однако для ряда задач, особенно связанных с серверными приложениями и техническим администрированием, использование консольного интерфейса может быть предпочтительнее.

CLI отличается лаконичностью, минимальными системными требованиями и эффективностью в сценариях удалённого управления, автоматизации и мониторинга. Он особенно подходит для приложений, работающих на удалённых серверах, где зачастую недоступны графические подсистемы (такие как GNOME, KDE и другие оконные окружения). Кроме того, консольный интерфейс позволяет отказаться от использования различных графических элементов управления (слайдеров, аккордеонов, выпадающих списков и прочих UI-компонентов), которые в данных сценариях не только избыточны, но и могут усложнять взаимодействие и увеличивать накладные расходы на разработку и поддержку.

В частности, при отображении состояния серверов UE консольный интерфейс обеспечивает быстрый доступ к чётко структурированной и

однозначной информации, значительно упрощая её обработку, чтение и понимание.

Таким образом, несмотря на очевидные ограничения в плане визуализации и интерактивности, консольный интерфейс становится оптимальным выбором в ситуациях, когда важнее простота, быстродействие и лёгкость интеграции с автоматизированными системами мониторинга и управления.

Взаимодействие с консолью можно реализовать самостоятельно, однако это зачастую неоправданно, поскольку существуют готовые решения, зарекомендовавшие себя в системных утилитах. Самым распространенным и легковесным программным инструментом такого типа является библиотека `ncurses/PDCurses` [12]. Библиотека `PDCurses` является аналогом библиотеки `ncurses`, которая была разработана специально под операционную систему Windows. Библиотеки имеют одинаковый API и имеют незначительные отличия, которые можно учесть для реализации кроссплатформенного ПО.

Перед началом использования `ncurses/PDCurses`, для начала необходимо собрать бинарные файлы библиотеки и подключить их в проект, чтобы линковщик понимал, откуда ему брать реализацию подключаемых заголовочных файлов. Процесс подключения библиотеки в проект полностью аналогичен процессу подключения библиотеки `Boost.Asio`.

Перед началом вывода какой-либо информации в консоль необходимо провести процесс инициализации и настройки библиотеки (листинг 16).

#### Листинг 16 – Инициализация `ncurses/PDCurses`

```
void ConsoleMonitoring::Init()
{
    // Configuring ncurses
    initscr();
    noecho();
    cbreak();
    curs_set(0);
    nodelay(stdscr, TRUE);
    keypad(stdscr, TRUE);

    start_color();
    init_pair(1, COLOR_BLACK, COLOR_GREEN);
    init_pair(2, COLOR_BLACK, COLOR_BLUE);
    StartDrawLoop();
}
```

Функция *initscr()* инициализирует библиотеку и создает стандартный экран *stdscr* для вывода данных; *cbreak()* включает режим ввода символов без буферизации, при котором символы становятся доступными программе сразу после их ввода, без ожидания нажатия клавиши Enter; *curs\_set(0)* скрывает курсор на экране; *nodelay(stdscr, TRUE)* делает функцию ввода символов *getch()* неблокирующей; *keypad(stdscr, TRUE)* включает поддержку обработки нажатия клавиш на клавиатуре; *init\_pair()* позволяет задавать любую пару (цветовую, шрифтовую) для форматирования вывода на экран и улучшения читаемости информации.

Метод *ConsoleMonitoring::StartDrawLoop()* содержит в себе логику отрисовки информации на экран, которая выполняется в бесконечном цикле (листинг 17).

#### Листинг 17 — Основной цикл вывода информации в консоль

```
void ConsoleMonitoring::StartDrawLoop()
{
    timeout(1000); // Ожидание ввода до 1 секунды
    while (true)
    {
        UpdateScreen();
        int ch = getch();
        if (ch == 265) // F1
        {
            ToggleSearchMode();
        }
        ...
        if (ch >= 32 && ch <= 126) // Ввод любого символа с клавиатуры
        {
            if (m_bIsSerching)
            {
                m_searchQueryString += static_cast<char>(ch);
            }
        }
    }
}
```

Функция *timeout(int)* приостанавливает цикл на заданное количество миллисекунд и ждет ввода символов с клавиатуры посредством функции *getch()*. Если символы не были введены, то программа спустя заданное количество миллисекунд продолжает свое выполнение. Это помогает снизить нагрузку на процессор и в то же время не вызывает никаких задержек в выполнении цикла

при вводе символов с клавиатуры. *ConsoleMonitoring::UpdateScreen()* представляет собой набор функций отрисовки и очистки (листинг 18).

#### Листинг 18 – Функция отрисовки информации на экран

```
void ConsoleMonitoring::UpdateScreen()
{
    clear();
    DrawTableHeader();
    DrawServers();
    DrawFooter();
    refresh();
}
```

Функция *clear()* очищает внутренний буфер (виртуальный экран), удаляя с него все содержимое. Важно, что функция не очищает текущее окно экрана напрямую, а прокручивает его, оставляя старую информацию вывода выше вне окна. По такому принципу прокрутки работает обновление экрана в большинстве приложений с консольным интерфейсом. *refresh()* переносит содержимое внутреннего буфера (виртуального экрана) на реальный терминал.

Для вывода сообщения на экран, например, заголовка таблицы серверов, который отображает колонки таблицы, предназначается функция *int mvprintw(int, int, const char \*, ...)*. Первым аргументом она принимает индекс колонки, вторым – индекс строки, и третьим – строку с сообщением, которое надо вывести в терминал, в C-стиле.

Для отрисовки заголовка таблицы, а именно для выделения его цветом, были использованы функции *attron(COLOR\_PAIR(1))* и *attroff(COLOR\_PAIR(1))*. Данные функции позволяют задать цвет заднего фота в терминале. Важно после отрисовки вызвать функцию *attroff(COLOR\_PAIR(1))* для того, чтобы вернуть цвет заливки консоли в значениям по умолчанию.

Отрисовка футера таблицы была реализована с использованием структур данных, входящих в STL (листинг 19).

#### Листинг 19 – Фрагмент программной реализации отрисовки футера

```
static const std::vector<std::pair<std::string, std::string>> keyMap =
{
    { "F1", "Search" },
    { "F2", "Change sort column" },
    { "F3", "Change sort direction" },
    { "F10", "Quit" }
};
```

```

for (auto& keyPair : keyMap)
{
    size_t keyLength = keyPair.first.length();
    size_t valueLength = keyPair.second.length();

    mvprintw(maxWindowHeight, (int)offset, keyPair.first.c_str());
    offset += keyLength;
    attron(COLOR_PAIR(2));

    mvprintw(maxWindowHeight, (int)offset, keyPair.second.c_str());
    attroff(COLOR_PAIR(2));
    offset += valueLength;
}

```

В листинге в структуре данных `std::vector<std::pair<std::string, std::string>>`, которая является аналогом структуры данных `std::map`, но с соблюдением заданного порядка ключей, записаны доступные опции и соответствующие им клавиши, с помощью которых их можно применить. К стандартным операциям работы с данными в таблицах является сортировка и фильтрация. Сортировка позволяет упорядочить нужные данные в заданном порядке. Фильтрация позволяет увидеть в списке только те данные, которые нужны пользователю или администратору, что значительно упрощает восприятие данных.

Для реализации сортировки данных также использовалась функция, входящая в STL, `std::sort()`, принимающая в качестве параметров итераторы и лямбда-функцию. Лямбда-функция определяет правила сортировки (листинг 20).

## Листинг 20 – Программная реализация сортировки списка серверов

```

std::sort(testServers.begin(), testServers.end(), [this](const
ServerInfo& A, const ServerInfo& B)
{
    switch (m_sortColumn)
    {
        case SortColumn::UUID:
            return m_bDesendingSort ? A.m_uuid > B.m_uuid : A.m_uuid
< B.m_uuid;
        case SortColumn::CURRENT_PLAYERS:
            return m_bDesendingSort ? A.m_currentPlayers >
B.m_currentPlayers : A.m_currentPlayers < B.m_currentPlayers;
        case SortColumn::MAX_PLAYERS:
            return m_bDesendingSort ? A.m_maxPlayers >
B.m_maxPlayers : A.m_maxPlayers < B.m_maxPlayers;
        case SortColumn::STATE:

```



Для реализации направления сортировки в поля класса необходимо было добавить две переменных *bool m\_bDesendingSort* и *SortColumn::m\_sortColumn*. Переменная *m\_bDesendingSort* представляет собой булеву переменную направления сортировки; переменная *m\_sortColumn* представляет собой класс перечисления, который содержит в себе текущую колонку сортировки данных.

Для реализации фильтрации данных использовались функции, входящие в STL: *vector::erase()* и *std::remove\_if()*. Метод *std::remove\_if()* возвращает новый итератор на границу между элементами, которые нужно оставить, и элементами, подлежащими удалению. *std::vector::erase()* физически удаляет перемещённые элементы из контейнера. Таким образом в контейнере, который необходимо отрисовать, остаются лишь те элементы, которые подходят под условие фильтрации.

В результате программной реализации механизма мониторинга вывод информации о запущенных выделенных серверах UE через консольный интерфейс представлен ниже (рисунок 21).

UUID	Address	[Current players]	Max players	State
test4321	127.0.0.1:7778	5	10	MATCH_STARTING
qwerty321123	127.0.0.1:7779	3	20	MATCH_IN_PROGRESS
test123	127.0.0.1:7777	1	10	LOBBY

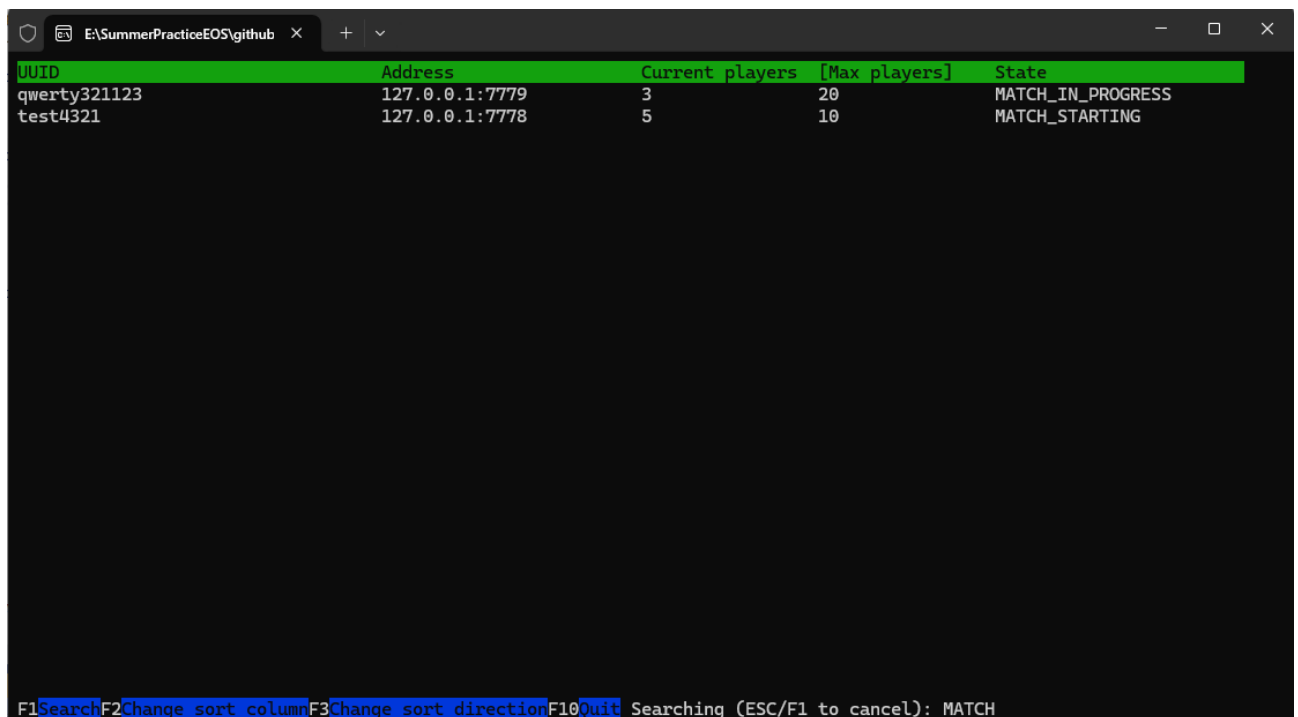
F1Search F2Change sort column F3Change sort direction F10Quit Sorting: Current players (Descending)

Рисунок 21 – Таблица мониторинга запущенных серверов в исполнении консольного интерфейса

На рисунке видно, что колонкой сортировкой по умолчанию в направлении возрастания является колонка *current\_players* – текущее количество подключенных пользователей. Данная колонка является наиболее релевантной,

так как по этому полю принимается решение о распределении пользователей между запущенными серверами. В футере представлены клавиши управления таблицей, по нажатию на которые выполняется операция, описанная справа от названия клавиши. Справа от описания клавиш находится название текущей выполняемой операции и ее параметры.

Для фильтрации данных необходимо нажать клавишу F1 и начать ввод значения, по которому необходимо произвести фильтрацию. Все данные обновляются в таблице по мере ввода в стандартный поток ввода (рисунок 22).



The screenshot shows a terminal window with a table of server data. The table has five columns: UUID, Address, Current players, [Max players], and State. There are two rows of data. Below the table, there is a search bar with a legend for function keys: F1Search, F2Change sort column, F3Change sort direction, and F10Quit. The search bar contains the text 'Searching (ESC/F1 to cancel): MATCH'.

UUID	Address	Current players	[Max players]	State
qwerty321123	127.0.0.1:7779	3	20	MATCH_IN_PROGRESS
test4321	127.0.0.1:7778	5	10	MATCH_STARTING

F1Search F2Change sort column F3Change sort direction F10Quit Searching (ESC/F1 to cancel): MATCH

Рисунок 22 – Применение фильтрации к таблице запущенных серверов

На рисунке продемонстрировано, что данные отображаются в соответствии с теми серверами, которые содержат в своем описании хотя бы одно упоминание подстроки “MATCH”. Кроме того, в правом нижнем углу отображается текущая введенная подстрока, по которой реализуется поиск. Для выхода из режима поиска необходимо нажать клавишу Escape или F1. Для закрытия инструмента мониторинга необходимо нажать клавишу F10.

Как уже было упомянуто ранее, отрисовка данных в терминал является блокирующей операцией. Прослушивание входящих данных по TCP-сокету также является блокирующей операцией. Для того, чтобы две операции запустились параллельно и выполняли свои задачи, необходимо разделить их

выполнение на два разных потока. Однако также необходимо соблюдать правило, чтобы точка входа программы (файл `main.cpp`) была максимально “чистая” и содержала в себе минимум логики. Для этого было дополнительно введен и разработан класс *Application*.

В конструкторе приложения можно настроить окружение программы, например чтение данных из конфига и их инициализация и использование (листинг 21)

#### Листинг 21 – Инициализация программы в классе приложения

```
Application::Application()
{
    m_tcpServer = boost::shared_ptr<TcpServer>(new TcpServer());
    m_consoleMonitoring = boost::shared_ptr<ConsoleMonitoring>(new
ConsoleMonitoring(m_tcpServer.get()));

    std::string logPath;
    ConfigHelper::ReadVariableFromConfig("appsettings.ini",
"Logger.logPath", logPath);
    m_logPath = logPath;
}
```

Достаточно важно упомянуть о использовании *boost::shared\_ptr<T>* вместо обычного указателя. Технология “умных” указателей создает накладные расходы, однако их использование в программе может быть оправдано. Во-первых, *shared\_ptr* автоматически управляет памятью и защищает от утечек в памяти. Во-вторых, *shared\_ptr* предотвращает раннее удаление и появление висячего указателя на объект. В-третьих, *shared\_ptr* делает передачу объекта между разными функциями и особенно потоками безопасным. Следовательно, использование *shared\_ptr* в данном случае оправдано. Запуск приложения был определен в методе класса `Application::Run()` (листинг 22).

#### Листинг 22 – Разделение программы на разные потоки

```
void Application::Run()
{
    try
    {
        Logger::GetInstance().SetLogFile(m_logPath);
        Logger::GetInstance() << "Starting applicaton..." <<
std::endl;
        boost::thread serverThread =
boost::thread(&TcpServer::StartServer, m_tcpServer.get());
        boost::thread consoleInterfaceThread =
boost::thread(&ConsoleMonitoring::Run, m_consoleMonitoring.get());
```

```

        serverThread.join();
        consoleInterfaceThread.join();
    }
    catch (std::exception ex)
    {
        Logger::GetInstance() << std::string(ex.what()) << std::endl;
    }
}

```

Функция *boost::thread::join()* присоединяет дочерний поток к основному потоку программы. Данный вызов гарантирует, что основной поток программы не завершится до тех пор, пока дочерний не закончит свое выполнение.

Таким образом точка входа приложения остается максимально “чистой” и не содержит в себе никакой дополнительной логики, кроме логики инициализации класса приложения и его запуска (листинг 23).

Листинг 23 – Точка входа приложения менеджера серверов

```

int main()
{
    Application application;
    application.Run();

    return 0;
}

```

### 4.3 Программная реализация механизма распределения пользователей по выделенным серверам

Механизм распределения пользователей по выделенным серверам реализован на уровне обработки входящих сообщений от клиентов. При поступлении данных от нового клиента сервер анализирует текущее состояние всех активных серверов UE, чтобы определить оптимальное место для подключения пользователя. Реализация включает ключевые этапы, описанные ниже.

Сначала проверяется наличие работающих серверов. Если список активных серверов пуст, сервер инициирует создание нового инстанса через программу демона с отправкой команды START. Новый клиент в этом случае классифицируется как инициатор запуска сервера.

Если работающие серверы уже существуют, осуществляется фильтрация тех из них, которые имеют свободные места для новых игроков, но при этом уже

содержат хотя бы одного подключённого пользователя. Это позволяет минимизировать количество частично заполненных серверов и более эффективно использовать ресурсы.

При наличии доступных серверов из их числа выбирается сервер с наибольшим числом игроков для дальнейшего подключения нового клиента. Такой подход способствует равномерной загрузке серверов и улучшает качество многопользовательского взаимодействия.

Если все существующие серверы заполнены, снова инициируется команда на создание нового выделенного сервера.

Для безопасной работы в многопоточной среде в критических местах обработки данных был использован механизм блокировок `std::mutex`, что позволило предотвратить гонки данных при доступе к общим структурам.

В многопоточных сетевых приложениях, таких как сервер на основе сетевой библиотеки `boost::asio`, одновременная обработка данных от множества клиентов является нормальной рабочей ситуацией. Каждый новый клиентский запрос может обрабатываться в отдельном потоке или через пул потоков, чтобы повысить производительность системы. В условиях многопоточности возникает одна из наиболее серьёзных проблем – гонка данных. Она проявляется тогда, когда два или более потока одновременно обращаются к одной и той же области памяти, при этом хотя бы один из них выполняет запись. Без надлежащей синхронизации это приводит к непредсказуемому поведению программы: повреждению данных, крахам, утечкам памяти или логическим ошибкам. Для устранения такой проблемы был применен механизм синхронизации `std::mutex` (листинг 24).

Листинг 24 – Применение механизма синхронизации при обработке запроса клиентов

```
void TcpServer::ProcessDataFromClient(std::string& message,
boost::shared_ptr<boost::asio::ip::tcp::socket> socket)
{
    Logger::GetInstance() << "Got data from client: " << message << std::endl;
    ClientInfo clientInfo;
    clientInfo.Socket = socket;
```

```

// Попытка захвата лока для _runningServers
std::lock_guard<std::mutex> lock(_runningServersMutex);

if (_runningServers.empty())
{
    return;
}

SendDataToSocket(socket, runningServerWithMostPlayers.m_URI);
}

```

В рассматриваемом листинге метод *ProcessDataFromClient* обращается к контейнеру *\_runningServers*, который хранит информацию о запущенных выделенных серверах. Поскольку одновременно несколько клиентов могут инициировать обработку своих запросов, возникает необходимость синхронизировать доступ к этому ресурсу. Например, в момент, когда один поток перебирает список серверов для поиска подходящего инстанса, другой поток мог бы изменить структуру этого списка, добавив, изменив или удалив сервер.

Для предотвращения подобных ситуаций используется механизм взаимного исключения. В методах дополнительно применяется обёртка `std::lock_guard<std::mutex>`, которая гарантирует, что доступ к *\_runningServers* возможен только одному потоку за раз. Благодаря этому достигается целостность данных и исключается риск гонок. Применение *lock\_guard* [13] делает код более безопасным, так как мьютекс автоматически освобождается при выходе объекта за пределы области видимости.

Дополнительно стоит отметить, что критическая секция внутри *ProcessDataFromClient* ограничена по времени: захват мьютекса осуществляется только на момент анализа состояния активных серверов и принятия решения о маршрутизации клиента. Это позволяет избежать ненужного блокирования других потоков и способствует лучшей масштабируемости системы при высокой нагрузке.

Также реализация учитывает возможность некорректного состояния списка серверов. Например, если один из инстансов стал недоступен между моментом проверки и попыткой подключения клиента, система фиксирует это с

помощью логирования и может инициировать удаление «битого» сервера из общего списка. Такой механизм, дополненный системой мониторинга и автоматического восстановления, повышает отказоустойчивость решения.

Наконец, модуль логирования (*Logger::GetInstance()*) играет ключевую роль в трассировке действий сервера: от получения клиентского сообщения до выбора конкретного инстанса. Это существенно облегчает диагностику, тестирование и эксплуатационную поддержку, особенно при отладке распределённых систем с большим числом одновременных подключений.

## ЗАКЛЮЧЕНИЕ

В ходе работы была разработана система автоматизированного запуска выделенных серверов UE, которая упрощает управление серверной инфраструктурой. Основой системы стал демон, запускающий экземпляры серверов, и менеджер, который обрабатывает запросы как от клиентов, так и от серверов, и взаимодействует с демоном.

Для обеспечения безопасности и удобства пользователей была реализована аутентификация через API транзакций EOS, включающая использование интерфейсов PurchaseInterface и StoreInterface. Это позволило организовать безопасный процесс аутентификации и управление покупками, предоставляя пользователям доступ к серверным услугам только при подтверждении их прав. Дополнительно был внедрен защищенный оффлайн доступ к данным о покупках, что дает пользователям возможность пользоваться системой даже при временных сбоях в подключении к сети.

Таким образом, разработанная система предоставляет функциональные возможности для запуска серверов в автоматизированном режиме; предоставляет информацию о каждом запущенном сервере и его загрузке через консольный интерфейс; безопасно управляет доступом через авторизацию на стороне клиента и поддерживает оффлайн-режим, что делает её эффективным и надёжным решением для многопользовательских приложений на базе UE.

Результаты работы демонстрируют практическую значимость предложенного решения для разработки современных онлайн-сервисов, требующих масштабируемой и управляемой серверной архитектуры. В дальнейшем систему можно адаптировать для облачных платформ и интеграции с другими сервисными экосистемами, расширяя её функциональность и повышая устойчивость. Полученные наработки могут стать основой для построения более сложных систем распределённого запуска серверов, применимых в индустрии видеоигр и других областях, где требуется динамическое управление серверными ресурсами.



## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Cross-Compiling for Linux // Unreal Engine Documentation [Электронный ресурс] – URL: <https://docs.unrealengine.com/4.26/en-US/SharingAndReleasing/Linux/GettingStarted/>. Дата обращения 14.02.2025.
2. TCP Socket Listener, Receive Binary Data From an IP/Port into UE4 // Unreal Community [Электронный ресурс] – URL: [https://unrealcommunity.wiki/tcp-socket-listener-receive-binary-data-from-an-ip/port-into-ue4-\(full-code-sample\)-1eefbvdk](https://unrealcommunity.wiki/tcp-socket-listener-receive-binary-data-from-an-ip/port-into-ue4-(full-code-sample)-1eefbvdk). Дата обращения 15.02.2025.
3. UE5 Multithreading With FRunnable And Threat Workflow // Algosyntax [Электронный ресурс] – URL: <https://store.algosyntax.com/tutorials/unreal-engine/ue5-multithreading-with-frunnable-and-thread-workflow/>. Дата обращения 15.02.2025.
4. Boost.Asio // Boost C++ Libraries [Электронный ресурс] – URL: [https://www.boost.org/doc/libs/1\\_76\\_0/doc/html/boost\\_asio.html](https://www.boost.org/doc/libs/1_76_0/doc/html/boost_asio.html). Дата обращения 21.02.2025.
5. Boost.Asio C++ Network Programming // Habr [Электронный ресурс] – URL: <https://habr.com/ru/articles/195794/>. Дата обращения 22.02.2025.
6. Процесс разработки и тестирования демонов // Habr [Электронный ресурс] – URL: <https://habr.com/ru/articles/108294/>. Дата обращения: 23.02.2025.
7. Larry L. Peterson, Bruce S. Davie Computer Networks: A Systems Approach. - 5 изд. - Morgan Kaufmann, 2011. - 920 с..
8. Паттерн Singleton // Habr [Электронный ресурс] – URL: <https://habr.com/ru/articles/147373/>. Дата обращения 01.03.2025.
9. Logger C++ // Habr [Электронный ресурс] – URL: <https://habr.com/ru/articles/838412/>. Дата обращения 01.03.2025.
10. Gettings started on Unix Variants // Boost C++ Libraries. [Электронный ресурс] – URL: [https://www.boost.org/doc/libs/1\\_86\\_0/more/getting\\_started/unix-variants.html](https://www.boost.org/doc/libs/1_86_0/more/getting_started/unix-variants.html). Дата обращения 09.03.2025.

11. Firewall-cmd // Debian [Электронный ресурс] – URL: <https://manpages.debian.org/testing/firewalld/firewall-cmd.1.en.html>.

Дата обращения 15.03.2025.

12. ncurses(3x) – Linux manual page // man7.org [Электронный ресурс] – URL: <https://man7.org/linux/man-pages/man3/ncurses.3x.html>. Дата обращения 18.03.2025.

13. std::lock\_guard // cppreference.com [Электронный ресурс] – URL: [https://en.cppreference.com/w/cpp/thread/lock\\_guard/](https://en.cppreference.com/w/cpp/thread/lock_guard/). Дата обращения 01.04.2025.

## **ПРИЛОЖЕНИЕ А. Графическая часть выпускной квалификационной работы**

В графическую часть выпускной квалификационной работы входят 8 чертежей.

- Редактирование виджета управления процессом запуска выделенного сервера на уровне редактора Unreal Engine.
- Конфигурационные файлы сервисов системы (конфигурационный файл клиента, конфигурационный файл демона, конфигурационный файл менеджера).
- Фрагменты лог-файла сервисов системы (фрагмент лог-файла программы менеджера, фрагмент лог-файла программы демона, фрагмент лог-файла запущенного сервера Unreal Engine).
- Файловая структура проекта менеджера и проекта демона.
- Процесс авторизации в режиме без доступа к Интернету.
- Виджет выбор режима игры в главном меню клиента.
- Блок-схема алгоритма обработки полученной от клиента команды.
- Блок-схема алгоритма чтения данных из сокета выделенного сервера Unreal Engine.