Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное 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 НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

НА ТЕМУ

«Разработка сетевой инфраструктуры многопользовательского приложения на Unreal Engine 4»

Студент РК6-21М — Д. В. Боженко
(Группа) (подпись, дата) (инициалы и фамилия)

Руководитель — Ф. А. Витюков
(подпись, дата) (инициалы и фамилия)

Москва, 2024 г.

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

УТВЕРЖДАЮ
Заведующий кафедрой РК6
А.П. Карпенко

«_____» _____ 2024 г.

ЗАДАНИЕ
на выполнение научно-исследовательской работы

по теме: Разработка сетевой инфраструктуры многопользовательского приложения на Unreal Engine 4

Студент группы РК6-21М

Боженко Дмитрий Владимирович
(Фамилия, имя, отчество)

Направленность НИР (учебная, исследовательская, практическая, производственная, др.) учебная
Источник тематики (кафедра, предприятие, НИР) кафедра

График выполнения НИР: 25% к 5 нед., 50% к 11 нед., 75% к 14 нед., 100% к 16 нед.

Техническое задание: проанализировать возможные варианты развертывания приложения на Unreal Engine с использованием Kubernetes и самописным менеджером серверов. Изучить возможные варианты построения программы менеджера серверов. Исследовать количество подключений к одной программе по одному порту для менеджера серверов.

Оформление научно-исследовательской работы:
Расчетно-пояснительная записка на 20 листах формата А4.
Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.):
_____
_____
_____
_____

Дата выдачи задания «15» февраля 2024 г.

Руководитель НИР _____ **Витюков Ф.А.**
(Подпись, дата) И.О. Фамилия

Студент _____ **Боженко Д. В.**
(Подпись, дата) И.О. Фамилия

Примечание: Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедр

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Развертывание приложения любого вида – это один из важных этапов разработки любого продукта. От качества развертывания приложения зависит такие параметры как отказоустойчивость, скорость и масштабируемость приложения, а также в целом дальнейшее качество его использования конечным пользователем.

Для того, чтобы развертывание приложения получилось качественным, необходимо провести предварительный анализ и расчет, а именно подсчитать максимально возможное количество одновременных подключений клиентов, максимальное количество требуемых вычислительных ресурсов, а также будет ли приложение использовать кластерную архитектуру или же использовать сервер в единственном экземпляре.

После проведенного анализа, используя подсчитанные данные, необходимо выбрать вид хостинга. Для больших проектов с серьезной нагрузкой речь идет о хостинге на физических выделенных серверах или же на виртуальных выделенных серверах.

Хотя развертывание на физическом аппаратном программном обеспечении на данный момент до сих популярно и в некоторых случаях незаменимо, большое количество крупных приложений разворачиваются и запускаются в облачных сервисах, так как там запуск приложения, как правило, более доступный и быстрый, чем настройка целого выделенного сервера. При этом хостинг приложения в облаке имеет лишь незначительные ограничения по сравнению с более привычным и традиционным способом развертывания.

В рамках данной работы была поставлена задача проанализировать возможные варианты создания менеджера серверов и реализовать один из них. Также необходимо изучить возможные способы запуска экземпляров приложения на Unreal Engine и программы менеджера серверов на Debian машине.

1. РАЗВЕРТЫВАНИЕ С ПОМОЩЬЮ ТЕХНОЛОГИИ КОНТЕЙНЕРИЗАЦИИ

Ранее в работе были рассмотрены технологии контейнеризации, технологии масштабирования и хостинга с помощью Kubernetes. Для того чтобы запустить любое приложение на одном из узлов кластера Kubernetes необходимо для начала контейнерезировать его. Существует два основных способа, которые позволяют запустить выделенный Unreal Engine сервер в Docker. Первый способ подразумевает использование подготовленного образа Linux-сервера, так как выделенный сервер проекта на Unreal Engine требует установленного в системе движка, который был собран из исходных кодов. Один из образов, расположенный 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 Unreal Engine. 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 инструментов 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).

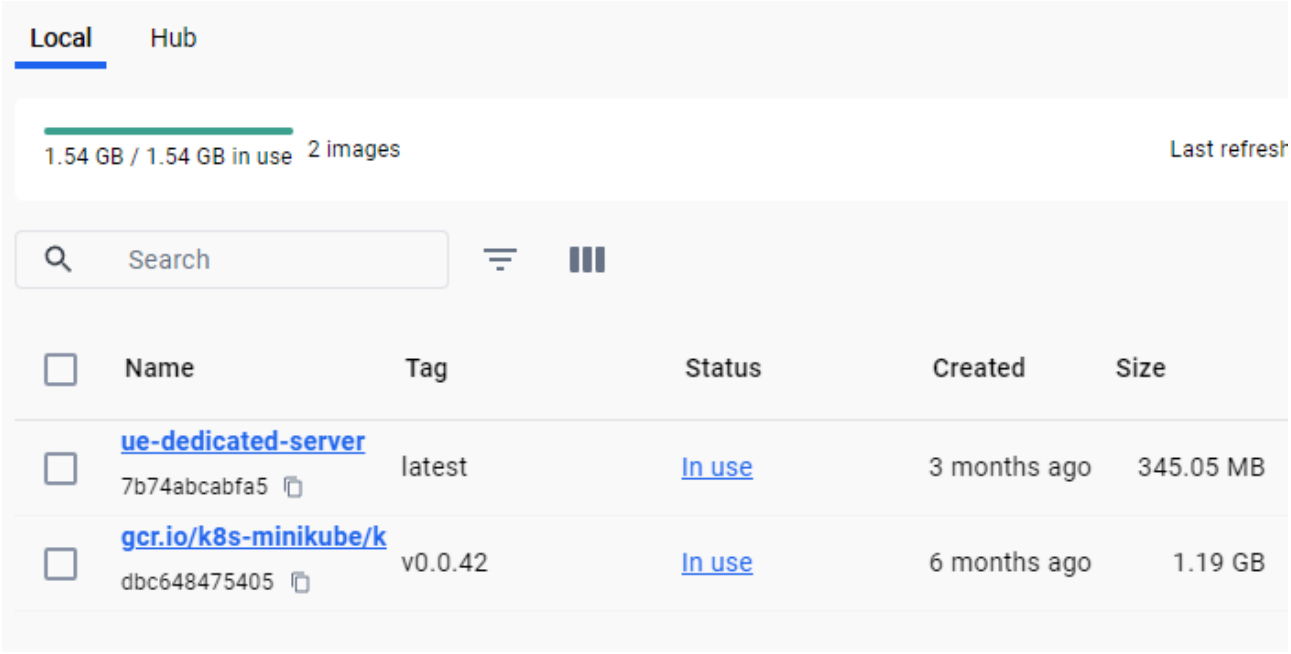

Рисунок 1. Размер созданных образов выделенного сервера и Minikube

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 N контейнеров, 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 цифра, естественно, будет больше.

Подводя итоги эксперимента с использованием технологии контейнеризации, можно сделать вывод, что Kubernetes, несомненно, является хорошим инструментов 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 Unreal Engine 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. РЕАЛИЗАЦИЯ МЕНЕДЖЕРА СЕРВЕРОВ

Главной идей написания собственного менеджера серверов (SM) 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 автоматическом режиме запускать на удаленном VPS или удаленной физической машине отдельные экземпляры выделенного сервера на Unreal Engine средствами сетевого программирования и программ-демонов.

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 со схемой реализации такой сетевой инфраструктуры.

Рисунок 2. Схема сетевой инфраструктуры с менеджером серверов

Анализируя рисунок 1, можно увидеть пример простой сетевой инфраструктуры, где имеется один узел SM и один узел VPS. У каждого клиента налажена двусторонняя связь с SM. Каждый клиент представляет собой приложение на Unreal Engine, где пользователь может изъявить желание создать новую сессию или присоединиться к уже существующей сессии. SM представляет собой программу, которая обрабатывает запросы клиентов и выделенных серверов приложения на Unreal Engine. Таким образом, к основным выполняемым задачам SM относятся следующие пункты:

1. Матчмейкинг. SM, обрабатывая запросы клиентов, должен заносить их в свою внутреннюю структуру данных и обеспечивать подбор игроков в соответствии с глобальной таблицей очков, которая принадлежит системе Epic Online Services (EOS). Таблица очков представляет собой некую оценку «мастерства» игрока, которая позволяет подбирать подходящих оппонентов для предстоящей сессии. EOS предоставляет API, с помощью которого можно определить текущее количество очков всех пользователей, зарегистрированных в системе.

2. Отправка команд. SM должен формировать запросы на создание или удаление экземпляров выделенных серверов приложения на Unreal Engine. Запросы передаются по сети программе-демону, которая обрабатывает полученные запросы и удаленно запускает/удаляет экземпляры выделанных серверов приложения на VPS или физической машине.

3. Мониторинг процессов. SM 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 SM.

Двусторонняя связь между клиентом и менеджером устроена следующем образом: когда клиент изъявляет желание создать сессию или присоединиться к существующей сессии, он делает сетевой запрос менеджеру серверов. Тот в свою

очередь обрабатывает запрос клиента и заносит его в структуру данных. После завершения процесса подбора игроков SM при наличии свободных VPS делает запрос программе-демону на удаленный запуск нового экземпляра выделенного сервера. При успешном запуске выделенного сервера тот посылает обратный запрос SM с информацией о его IP-адресе и порте, на котором запустился процесс. SM обрабатывает данную информацию и отправляет запрос всем клиентам, которые изъявили желание начать сессию в соответствии с системой подбора игроков. Клиенты, получив информацию, подключаются по указанному IP-адресу и порту в единое виртуальное пространство.

Существует множество инструментов, которые позволяют реализовать SM. В рамках данной работы были рассмотрены варианты создания SM с помощью движка Unreal Engine и сетевой библиотеки Boost.Asio.

2.1. Реализация на Unreal Engine

Unreal Engine 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 1 целесообразно использовать транспортный протокол TCP, так как он обеспечивает надежную отправку данных в заданном порядке.

Для обмена данными по сети в SM необходимо использовать два вида сокета [2]: сокет для прослушки входящих соединений (LS) и сокет подключений для отправки данных на клиент или программе демону или обратно. Для хранения сокета в Unreal Engine используется класс *FSocket*. Инициализация LS по протоколу TCP представлена ниже.

Листинг 1. Инициализация сокета для прослушки соединений

```
FIPv4Endpoint Endpoint(outAddress, Port);
ListenerSocket = FTcpSocketBuilder("ServerListeningSocket")
     .AsReusable()
     .BoundToEndpoint(Endpoint)
     .Listening(8);

TcpListener = MakeShareable(new FTcpListener(*ListenerSocket));
```

Анализируя листинг 1 можно увидеть, что LS создается с помощью конструктора класса *FTcpSocketBuilder*. Для создания сокета необходимо указать его имя, а также *Endpoint* – IP-адрес и порт, по которому SM 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. Дале на основе LS создается новый экземпляр класса *FTcpListener* и оборачивается в класс *TSharedPointer<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.

Листинг 2. Выделение дочернего потока под прослушивание соединений

```
if (TcpListener->Init())
{
     GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Green,
     FString::Printf(TEXT("Start listening on %s:%d"), *Adress,
     Port));
     Thread = FRunnableThread::Create(TcpListener.Get(),
     TEXT("TCP_Subsystem_Thread"), 0, TPri_BelowNormal);
}
```

Анализируя листинг 2, можно увидеть, что с помощью статического метода *FRunnableThread::Create(FRunnable*)* в переменную Thread 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*, структура которого будет приведена далее.

Для того, что создать сокет подключения, необходимо в метод *FTcpListener::OnConnectionAccepted* передать колбэк-фукнцию вида

OnConnected(FSocket ClientSocket, const FIPv4Endpoint& ClientEndpoint)*, в теле которой необходимо записывать сокет подключения в переменную, чтобы получать и отправлять сообщения.

Листинг 3. Выделение дочернего потока под сокет подключённого клиента

```
if (!ConnectionSocket)
{
	ConnectionSocket = ClientSocket;
	UE_LOG(LogTemp, Log, TEXT("New client connected from
	%s"),*ClientEndpoint.ToString())
	FString confirmMessage = FString::Printf(TEXT("Connected to
	%s"), *(FApp::GetName()));

	ReceiveThread = FRunnableThread::Create(new
	FReceiveThread(ConnectionSocket), TEXT("ReceiveThread"), 0,
	TPri_BelowNormal);
	ConnectionSocket = nullptr;
}
```

В листинге 3 представлен фрагмент кода, где с помощью уже упомянутого статического метода *FRunnable::Create* порождается дочерний поток для выполнения логики, содержащейся в экземпляре класса *FReceiveThread*.

Класс *FReceiveThread* представляет собой класс, унаследованный от класса *FRunnable*. Ниже представлена его структура.

Листинг 4. Структура класса FRunnable

```
class SERVERMANAGER_API FReceiveThread: public FRunnable
{
public:
	virtual bool Init() override
	{
	}
	virtual uint32 Run() override
	{
		return 0;
	}
	virtual void Stop() override
	{
	}
	virtual void Exit() override
	{
	}
private:
	FSocket* ClientSocket;
	bool bIsStopped;
```

Анализируя листинг 4, можно увидеть 4 основных функции, которые надо переопределить, чтобы иметь возможность запускать логику работы с сокетом подключения в дочернем потоке [3]. Основной функцией, которую надо переопределить является функция *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 основой цикл успешно выполнил свою работу. При завершении цикла порожденный дочерний поток удаляется. В функции *void Exit()* опционально можно поместить логику по очистке.

Таким образом, создав два вида сокета, SM может выполнять базовый функционал, заключающийся в принятии и передачи сообщений по сети с использованием транспортного протокола TCP. Основным минусом реализации SM или программы-демона является то, что SM реализованный на Unreal Engine так же потребляет довольно много оперативной памяти, так как движок в режиме реального времени затрачивает вычислительные ресурсы на задачи, не связанные с сетевым обменов данных. В простой базовой версии, где реализовано прослушивание входящих соединений и моментальный ответ на входящий запрос, SM использует 400 Мб оперативной памяти, что довольно много для приложения с таким простым сетевым функционалом.

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

2.2. Реализация на 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 ней не обязывает разработчика создавать .lib или .dll файлы. Достаточно лишь добавить дополнительный каталог включаемых файлов в настройки C++ проекта (Рисунок 3).

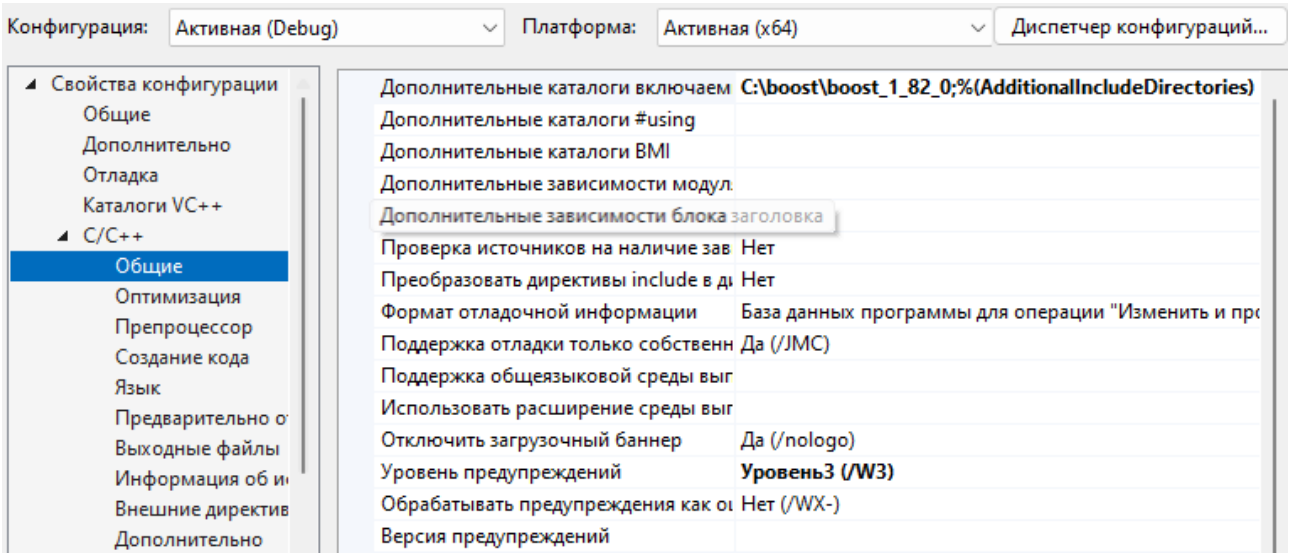

Рисунок 3. Подключение библиотеки Boost к проекту C++

Реализация SM на Boost.Asio аналогична реализации на Unreal Engine. В программе должны быть два вида сокетов, один из которых существует на протяжении всей жизни приложения и прослушивает входящие соединения. Второй вид сокета создается при подключении клиента и позволяет реализовать через себя передачу сообщений по сети. Сервер будет представлять собой многопоточную синхронную реализацию, так как выполнение синхронных операций более безопасно.

Для создания LS также, как и в Unreal указать структуру подключения Endpoint и запустить цикл прослушивания [5].

Листинг 5. Создание сокета прослушки входящих соединений

```
// Инициализация эндпоинта хоста, куда будут подключаться клиенты
Endpoint endpoint(ip::address::from_string("127.0.0.1"), 8870);
// Инициализация сокета для прослушки входящих соединений
tcp::acceptor acceptor(context, endpoint);
```

Как видно из листинга 5, создание LS на основе протокола TCP реализуется довольно легко. Прослушивание также запускается в бесконечном цикле (листинг 6).

Листинг 6. Основной цикл программы на Bost.Asio

```
while (true)
{
    boost::this_thread::sleep(1);
    socketPtr clientSocket(new tcp::socket(context));
    acceptor.accept(*clientSocket);
    boost::thread(boost::bind(readDataFromClient, clientSocket));
}
```

На листинге видно, что в бесконечном цикле запускается функция прослушивания входящих соединений по созданному сокету. Как и в Unreal Engine, 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 SM и основной поток выполнения программы не должен быть заблокирован.

На стороне клиента (приложение на Unreal Engine) необходимо реализовать возможность прием/передачу сообщений по сети с помощью протокола TCP, как описано в п. 2.1. Для отправки сообщений был реализован простой виджет, с двумя управляющими элементами (Рисунок 4.).

Рисунок 4. Виджет на клиенте для отправки сообщений в менеджер серверов

По нажатии на элемент виджета “Find match” в SM по указанному адресу отправляется сообщение в виде обычной строки о том, что пользователь желает создать сессию или подключиться к существующей (Рисунок 5).

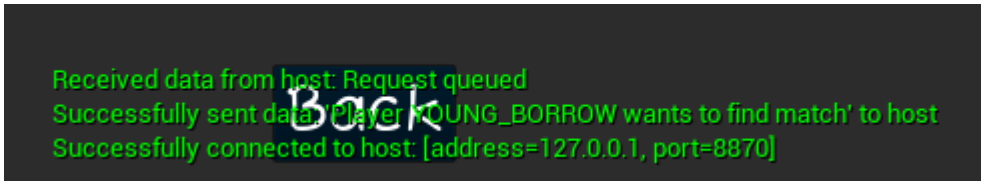

Рисунок 5. Сообщения отладки об успешном подключении и успешной отправки сообщения в менеджер серверов

SM прослушивает соединения по порту 8870, обрабатывает входящие запросы и выводит отладочное сообщение в консоль (Рисунок 6).

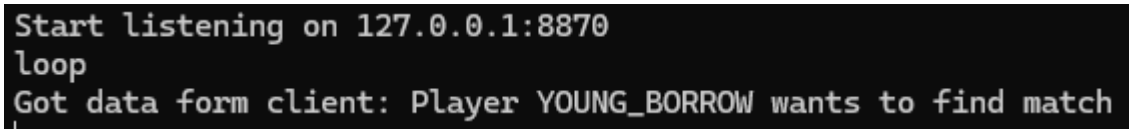

Рисунок 6. Сообщения отладки об успешном получении данных от клиента

Таким образом реализованный с помощью сетевой библиотеки для C++ Boost.Asio менеджер серверов взаимодействует с клиентами

(приложениями на Unreal Engine) по локальной сети с использованием транспортного протокола TCP.

3. ЗАПУСК МЕНДЕЖЕРА СЕРВЕРОВ НА LINUX-СЕРВЕРЕ

В действительности программа менеджера серверов должна быть запущена 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.

Для запуска программы менеджера серверов 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.

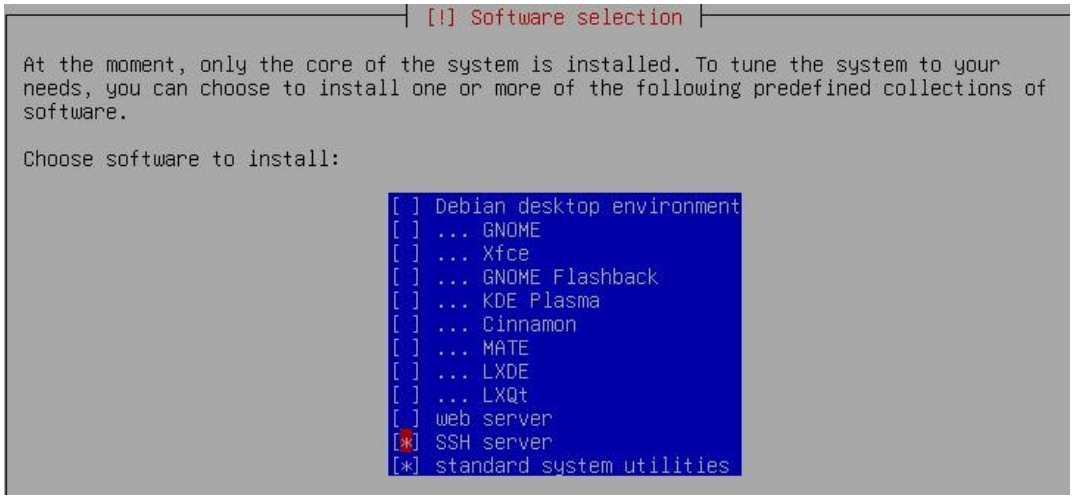

Рисунок 7. Установка программного обеспечения для Debian

Анализируя рисунок 7, можно увидеть, что из всех опций программного обеспечения была выбрана опция установки ssh-сервера и стандартных системных утилит без графического интерфейса.

Для того, чтобы запустить программу менеджера серверов на Linux-сервера также необходимо скачать библиотеку boost и провести ее установку. 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 ServerManager.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; *SeverManager.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 также была запущена программа менеджера серверов.

```
oost/ServerManager/ServerManager# g++ -I /home/user/boost/boost_1_82_0 ServerMan
ager.cpp -L /home/user/boost/boost_1_82_0/stage/lib -lboost_thread -lboost_syste
m
root@debserver:/home/user/server-manager/bmstu_src/Master/Diploma/ServerManagerB
oost/ServerManager/ServerManager# ./a.out
Start listening on 127.0.0.1:8870
```

Рисунок 8. Результаты запуска программы менеджера серверов на дистрибутиве Debian

ЗАКЛЮЧЕНИЕ

В результате выполнения исследовательской работы было проверено, что средства контейнеризации обладают значительными накладными расходами по памяти и могут не подходить для запуска небольших приложений. Также в результате выполнения работы было проведено исследование вариантов реализации сетевой инфраструктуры приложения на Unreal Engine. В качестве одного из вариантов было выбрана самостоятельная реализация сетевой инфраструктуры без использования технологии контейнеризации с помощью программы менеджера серверов и программы демона. Также был выполнен эксперимент, в котором выбирался инструмент реализации программы менеджера серверов. Результаты эксперимента показали, что наиболее эффективным по памяти является вариант разработки сервера менеджеров и остальных участников сетевой инфраструктуры с помощью сетевой библиотеки Boost.Asio на C++, которая будет иметь только консольный интерфейс для взаимодействия с ней.

В результате практической части работы была реализована программа менеджера серверов, налажено взаимодействие по транспортному протоколу TCP между программой менеджера серверов и программой клиентом на Unreal Engine. Также была выполнена настройка и запуск программы на Linux-сервере дистрибутива Debian.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Cross-Compiling for Linux. Unreal Engine Documentation [Электронный ресурс] // Режим доступа: https://docs.unrealengine.com/4.26/en-US/SharingAndReleasing/Linux/GettingStarted/ (дата обращения 15.02.2024);

2. TCP Socket Listener, Receive Binary Data From an IP/Port into UE4. Unreal Community [Электронный ресурс] // Режим доступа: https://unrealcommunity.wiki/tcp-socket-listener-receive-binary-data-from-an-ip/port-into-ue4-(full-code-sample)-1eefbvdk (дата обращения 20.03.2024);

3. UE5 Multithreading With FRunnable And Threat Workflow. Algosyntax [Электронный ресурс] // Режим доступа: https://store.algosyntax.com/tutorials/unreal-engine/ue5-multithreading-with-frunnable-and-thread-workflow/ (дата обращения 25.03.2024);

4. Boost.Asio. Boost C++ Libraries [Электронный ресурс] // Режим доступа: https://www.boost.org/doc/libs/1_76_0/doc/html/boost_asio.html (дата обращения 10.04.2024);

5. «Boost.Asio C++ Network Programming». Habr [Электронный ресурс] // Режим доступа: https://habr.com/ru/articles/195794/ (дата обращения 15.04.2024);