



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

ОТЧЕТ ПО ПРОИЗВОДСТВЕННОЙ ПРАКТИКЕ

Студент Боженко Дмитрий Владимирович
Группа РК6-21М
Тип практики Эксплуатационная
Название предприятия «НИИ Автоматизации Производственных
Процессов МГТУ им. Н.Э. Баумана»

Студент	_____	<u>Боженко Д.В.</u>
	<i>подпись, дата</i>	<i>фамилия, и.о.</i>
Руководитель практики	_____	<u>Витюков Ф. А.</u>
	<i>подпись, дата</i>	<i>фамилия, и.о.</i>

Оценка: _____

Москва, 2024 г.

СОДЕРЖАНИЕ

Введение	4
1. Программа-демон	5
1.1. Программная реализация	5
1.2. Запуск программы на Linux Debian	8
2. Обратное сетевое взаимодействие	13
2.1. Программная реализация	14
2.2. Обработка сообщений на клиенте	18
2.3. Программная реализация логгера	20
Заключение	21
Список используемых источников	22

ВВЕДЕНИЕ

Одним из важнейших аспектов работы с Unreal Engine 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.

В рамках данной практической работы рассматривается разработка и внедрение системы автоматического запуска выделенных серверов Unreal Engine на основе C++. Разработанная система позволяет пользователям инициировать запуск серверов по запросу с клиентского приложения. Такая автоматизация значительно упрощает управление серверами, снижает нагрузку на операционные команды и позволяет оптимизировать использование вычислительных ресурсов.

1. ПРОГРАММА-ДЕМОН

Программа-демон (демон) – это программа в UNIX-подобных системах, которая запускается в фоновом режиме и выполняет поставленные ей задачи. К основным задачам демона в рамках данной практической работы можно отнести умение принимать команды и реагировать на них; умение запускать по команде экземпляр выделенного сервера Unreal Engine; умение собирать информацию системы о занятых и свободных вычислительных ресурсах; умение собирать информацию о состоянии всех запущенных выделенных серверов Unreal Engine.

1.1. Программная реализация

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. Прослушивание по сокету является блокирующей операцией из-за наличия бесконечного цикла, поэтому необходимо было создать отдельный поток и инициализировать в нем сокет, как показано в листинге ниже.

Листинг 1. Создание сокета, принимающего входящие сообщения

```
void createAcceptThread()
{
    const unsigned int port = 8871;
    // Создание контекста
    boost::asio::io_context context;
    boost::asio::ip::tcp::endpoint endpoint(boost::asio::ip::tcp::v4(),
port);

    // Инициализация сокета для прослушки входящих соединений
    boost::asio::ip::tcp::acceptor acceptor(context, endpoint);
    std::cout << "Start listening on 0.0.0.0:" << port << std::endl;

    while (true)
    {
        boost::this_thread::sleep(boost::posix_time::seconds(1));
        boost::shared_ptr<boost::asio::ip::tcp::socket> clientSocket(new
boost::asio::ip::tcp::socket(context));
        acceptor.accept(*clientSocket);
        boost::thread(boost::bind(handleIncomeQuery, clientSocket));
    }
}
```

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 функции `handleIncomeQuery`, которая читает сообщения из сокета `clientSocket`, создаваемого для каждого входящего подключения.

Листинг 2. Функция обработки входящих команд

```
void handleIncomeQuery(boost::shared_ptr<boost::asio::ip::tcp::socket>
socket)
{
    bool bIsReading(true);
    while (bIsReading)
    {
        char data[512];

        size_t bytesRead = socket->read_some(boost::asio::buffer(data));
        if (bytesRead > 0)
        {
            const std::string message = std::string(data, bytesRead);
            std::cout << "Handle income query: " << message << std::endl;
            if (message == "Start")
            {
                std::cout << "Starting server instance..." << std::endl;
                boost::thread(startServerInstance).detach();
            }
            bIsReading = false;
        }
    }
}
```

В листинге 2 можно увидеть пример, что при успешном чтении данных из `clientSocket` и получении команды “*Start*”, вызывается функция `startServerInstance`. Важно отметить, что вызов функции является блокирующей операций. 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. Функция `startServerInstance` представлена ниже.

Листинг 3. Функция запуска экземпляра сервера

```
void startServerInstance()
{
    #ifdef _WIN32
        const std::string scriptPath =
"E:\\Master\\sem2\\MMAPS\\Releases\\WindowsServer\\Lab4ServerPackaged.bat";
    #else
        const std::string scriptPath = "/home/user/dedicated-
server/LinuxServer/Lab4Server.sh";
    #endif // _WIN32
    try
    {
        boost::process::child childThreat(scriptPath,
boost::process::std_out > stdout, boost::process::std_err > stderr);
        childThreat.wait();
        if (childThreat.exit_code() == 0)
        {
            std::cout << "Server instance started successfully!" <<
std::endl;
        }
        else
        {
            std::cerr << "Error, while starting server instance. Exit code: "
<< childThreat.exit_code() << std::endl;
        }
    }
    catch (const std::exception& exception)
    {
        std::cerr << "Exception occured, while starting server instance: "
<< exception.what() << std::endl;
    }
}
```

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 поток вывода ошибок для дальнейшей отладки.

1.2. Запуск программы на Linux Debian

Для запуска программы на Linux Debian для начала необходимо установить чистый дистрибутив без какого-либо графического интерфейса. Для более простой реализации в рамках разработки было принято решения использовать гипервизор VirtualBox. Для того, чтобы виртуальная машина была видна во внутренней сети, необходимо провести первоначальные сетевые настройки гипервизора.

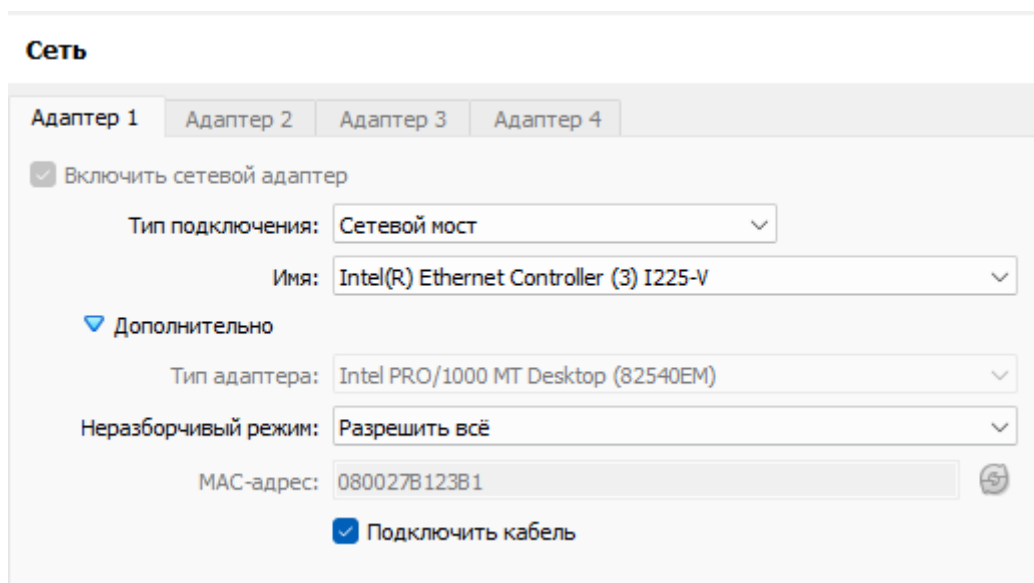


Рисунок 1. Сетевые настройки гипервизора

Как видно на рисунке 1, 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*.


```

user@debserver-daemon:~/daemon/bmstu_src/Master/Diploma/DaemonBoost/DaemonBoost/DaemonBoost$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host noprefixroute
        valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:b1:23:b1 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.12/24 brd 192.168.1.255 scope global dynamic enp0s3
        valid_lft 84546sec preferred_lft 84546sec
    inet6 2a00:1370:81a0:3b89:a00:27ff:feb1:23b1/64 scope global dynamic mngtmpaddr
        valid_lft 481sec preferred_lft 481sec
    inet6 fe80::a00:27ff:feb1:23b1/64 scope link
        valid_lft forever preferred_lft forever
user@debserver-daemon:~/daemon/bmstu_src/Master/Diploma/DaemonBoost/DaemonBoost/DaemonBoost$ _

```

Рисунок 2. Вывод сетевых параметров VM

Чтобы понять, что VM доступна в локальной сети и между основной машиной и VM доступны сетевые взаимодействия, была выполнена команда *ping 192.168.1.12* с основной машины.

```

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 3. Выполнение проверки на доступность VM в локальной сети

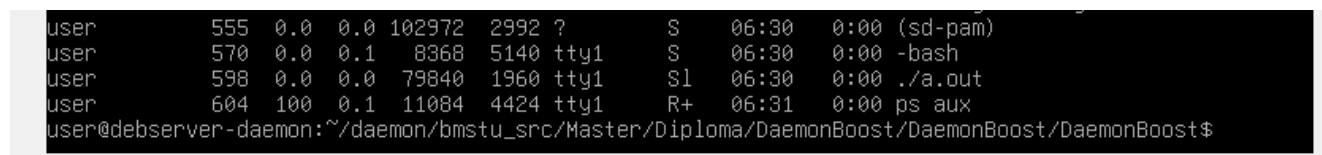
Далее, чтобы скомпилировать программу для начала была скачена и установлена библиотека *boost*. Для этого была использована команда *wget* для скачивания архива; для его разархивации была использована команда *tar -C /home/user/boost. /home/user/boost* – рабочая директория, где в дальнейшем будут лежать исходные и бинарные файлы библиотеки. После установки для компиляции программы необходимо в директории, где лежит программа, выполнить команду

```

g++ -I /home/user/boost/boost_1_82_0 DaemonBoost.cpp -L
/home/user/boost/boost_1_82_0/stage/lib -lboost_filesystem -lboost_system -
lboost_thread [1]

```

и запустить 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>`.



```
user      555  0.0  0.0 102972 2992 ?        S   06:30   0:00 (sd-pam)
user      570  0.0  0.1   8368 5140 tty1    S   06:30   0:00 -bash
user      598  0.0  0.0   79840 1960 tty1    Sl  06:30   0:00 ./a.out
user      604  100  0.1  11084 4424 tty1    R+  06:31   0:00 ps aux
user@debsserver-daemon:~/daemon/bmstu_src/Master/Diploma/DaemonBoost/DaemonBoost/DaemonBoost$
```

Рисунок 4. Просмотр работающего процесса через команду `ps aux`

Далее для того, чтобы из локальной сети можно было подключиться к процессу демона, необходимо изменить правила *firewall* и разрешить прослушивание по указанному порту и протоколу. Для открытия порта была использована программная утилита *firewalld* [2]. Ниже представлен набор команд, который использовался для открытия порта.

Листинг 4. Команды 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.

```

user@debsserver-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@debsserver-daemon:~/daemon/bmstu_src/Master/Diploma/DaemonBoost/DaemonBoost/DaemonBoost$ _

```

Рисунок 5. Проверка всех сервисов, прослушивающих входящие соединения по протоколам *TCP/UDP*

После выполнения отладки запуска сервиса было выполнено копирование файлов выделенного сервера под ядро UNIX с основной машины, на виртуальную машину Linux Debian. Было принято решение передать файлы 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 Unreal Engine 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.

```

После выполнения настроек и установок была проверена работоспособность системы. В результате обработки двух команд “Start” удаленно были запущены два экземпляра выделенных серверов, как показано на рисунке ниже.

```

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:*
udp      0      0 0.0.0.0:7778            0.0.0.0:*
udp      0      0 0.0.0.0:68              0.0.0.0:*
user@debsserver-daemon:~/daemon/bmstu_src/Master/Diploma/DaemonBoost/DaemonBoost/DaemonBoost$ _

```

Рисунок 6. Список запущенных выделенных серверов Unreal Engine

На рисунке 6 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 выделенных серверов Unreal Engine, 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 Unreal Engine на одной машине. Также с помощью команды *ps aux* была проверен корректный запуск процессов Unreal Engine.

```
user      650  0.0  0.0   2576   908 tty1      S   07:12   0:00 /bin/sh /home/user/dedicated-serv
user      657  6.6  5.2 1679872 209752 tty1    Sl  07:12   0:03 /home/user/dedicated-server/Linux
user      687  0.0  0.0   2576   936 tty1      S   07:12   0:00 /bin/sh /home/user/dedicated-serv
user      694  8.2  5.2 1679944 208680 tty1    Sl  07:12   0:02 /home/user/dedicated-server/Linux
user      722  0.0  0.1  11084   4328 tty1      R+  07:13   0:00 ps aux
user@debserver-daemon:~/daemon/bmstu_src/Master/Diploma/DaemonBoost/DaemonBoost/DaemonBoost$ _
```

Рисунок 7. Список запущенных процессов серверов Unreal Engine

Для проверки доступности запущенных серверов в рамках отладки с клиента Unreal Engine была выполнена команда *open 192.168.1.12:777* и *192.168.1.12:7778*, которая позволила подключиться к серверам Unreal Engine, запущенным на VM Linux Debian.

2. ОБРАТНОЕ СЕТЕВОЕ ВЗАИМОДЕЙСТВИЕ

Под обратным сетевым взаимодействием подразумевается отправка сообщения с запущенного экземпляра выделенного сервера на клиент Unreal Engine. 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 8).

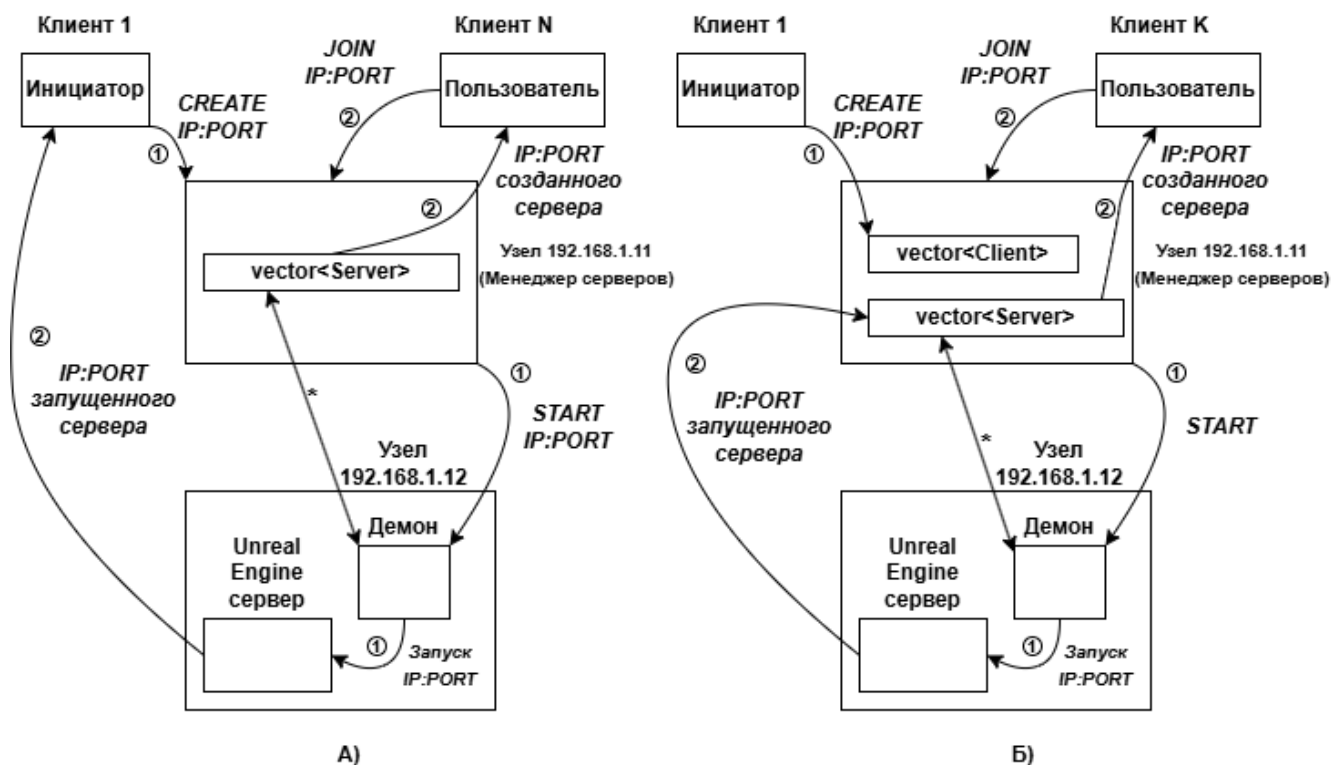


Рисунок 8. А) Схема обратного сетевого взаимодействия напрямую с клиентом.

Б) Схема обратного сетевого взаимодействия через менеджера серверов

Анализируя рисунок 8 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 Unreal Engine.

Во втором подходе сообщение с IP-адресом и портом передается обратно менеджеру серверов, 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 серверов. К минусам такого подхода можно отнести большее 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.

2.1. Программная реализация

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 запуска приложение – заранее никогда не известен и должен быть получен программно не из конфига (листинг 5).

Листинг 5. Программное получение порта, на котором запущен сервер

```
FString LocalNetworkAddress = NetDriver->LowLevelGetNetworkNumber();
if (LocalNetworkAddress.IsEmpty())
{
    UE_LOG(LogTemp, Error, TEXT("NetworkAddressString is empty"));
    return;
}
FString AdressString;
FString PortString;
if (!LocalNetworkAddress.Split(TEXT(":"), &AdressString, &PortString))
{
    UE_LOG(LogTemp, Error, TEXT("Failed to get port"));
    return;
}
int32 Port = FString::Atoi(*PortString);
UE_LOG(LogTemp, Log, TEXT("Server is listening on port: %d"), Port);
SendUriToServerManager(Port);
```

Анализируя листинг 5, можно увидеть, что 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 серверов была записана в конфиг приложения Unreal Engine.

В программе сервера менеджеров 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 6).

Листинг 6. Обработка адреса запущенного выделенного сервера

```
void TcpServer::ProcessDataFromDaemon(std::string& message)
{
    // Обработка присланного URI от DedicatedServer
    std::cout << "Got URI form started DedicatedServer: " << message <<
std::endl;

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

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

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

Анализируя листинг 6, 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 сетевого взаимодействия клиента-инициатора с системой (Рисунок 9).

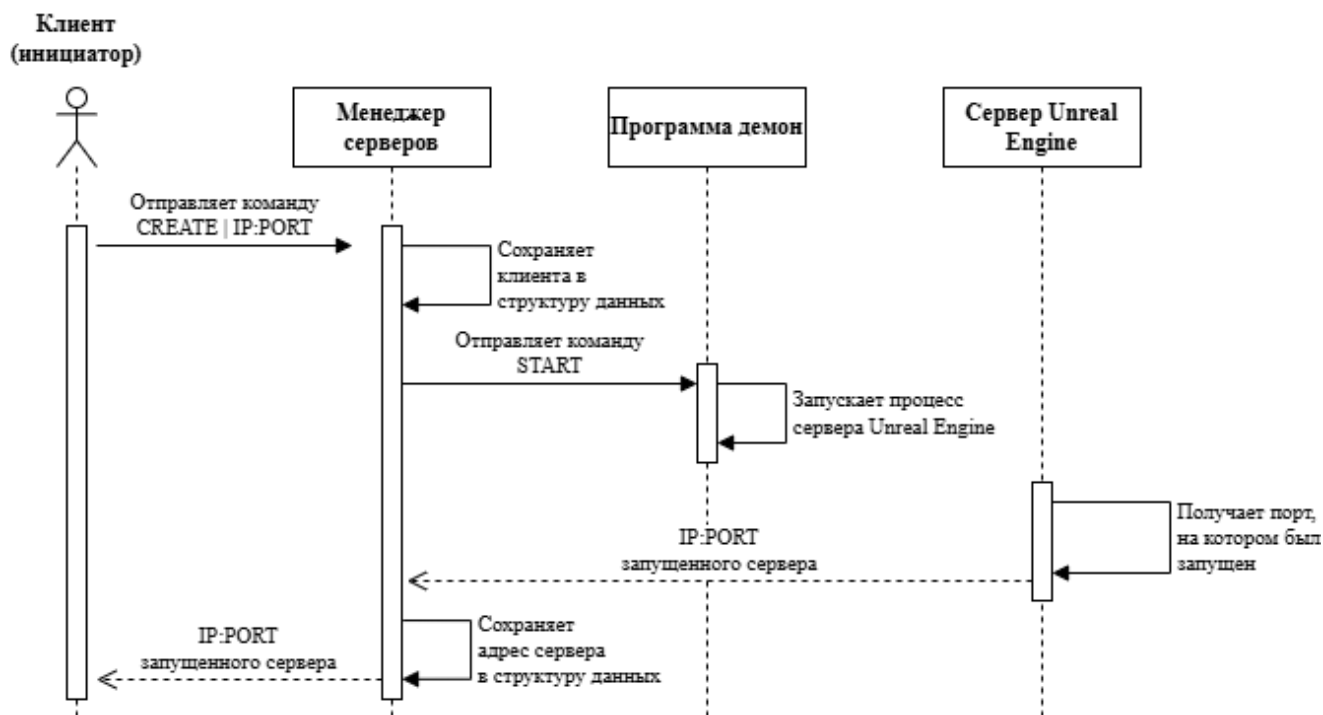


Рисунок 9. UML-диаграмма последовательности взаимодействия клиента-инициатора с системой

В итоге вышеописанного взаимодействия в логах программы менеджера серверов появляются следующие логи (Рисунок 10).

```

[2024-08-26 22:15:21] Start listening on 0.0.0.0:8870

Got data from client: CREATE
Sending command to daemon: Start
Added client to queue: [CLIENT_ADDRESS=127.0.0.1:54578,CLIENT_TYPE=INITIATOR]
Got URI form started DedicatedServer: 127.0.0.1:7777
Senging data to cleint: 127.0.0.1:7777
  
```

Рисунок 10. Логи программы менеджера серверов после запуска выделенного сервера

В логах программы-демона фиксируются события запуска экземпляров выделенных серверов и ошибки, если они возникают во время обработки команд запуска (Рисунок 11).

```

Start listening on 0.0.0.0:8871
Handle income query: Start
Starting server instance...

E:\SummerPracticeEOS\github\bmstu_src\bmstu_src\Master\Diploma\DaemonBoost\DaemonBoost\Daemo
ticeEOS\github\bmstu_src\bmstu_src\Master\Diploma\Client\Binaries\Win64\Lab4Server.exe -log
Server instance started successfully!

```

Рисунок 11. Логи программы-демона после запуска выделенного сервера

Логи выделенного сервера Unreal Engine предоставляют информацию о том, куда и с каким статусом отправилось сообщение о порте после запуска (Рисунок 12).

```

[2024.08.26-19.18.23:443][ 0]LogLoad: Game class is 'BP_EmptyLobbyGameMode_C'
[2024.08.26-19.18.23:445][ 0]LogNet: ReplicationDriverClass is null! Not using ReplicationDriver.
[2024.08.26-19.18.23:445][ 0]LogNetCore: DDoS detection status: detection enabled: 0 analytics enabled: 0
[2024.08.26-19.18.23:446][ 0]LogInit: WinSock: Socket queue. Rx: 131072 (config 131072) Tx: 131072 (config 131072)
[2024.08.26-19.18.23:446][ 0]LogNet: Created socket for bind address: 0.0.0.0 on port 7777
[2024.08.26-19.18.23:452][ 0]PacketHandlerLog: Loaded PacketHandler component: Engine.EngineHandlerComponentFactory
[2024.08.26-19.18.23:481][ 0]LogNet: GameNetDriver IpNetDriver_2147482566 IpNetDriver listening on port 7777
[2024.08.26-19.18.23:499][ 0]LogWorld: Bringing World /Game/Maps/LobbyMap.LobbyMap up for play (max tick rate 30) at
[2024.08.26-19.18.23:499][ 0]LogOnline: Warning: OSS: Unable to AutoLogin user (0) due to missing auth command line
[2024.08.26-19.18.23:503][ 0]LogWorld: Bringing up level for play took: 0.021968
[2024.08.26-19.18.23:514][ 0]LogGameMode: Display: Match State Changed from EnteringMap to WaitingToStart
[2024.08.26-19.18.23:515][ 0]LogTemp: Server is listening on port: 7777
[2024.08.26-19.18.23:519][ 0]LogTemp: Successfully created socket to host: [address=127.0.0.1, port=8870]
[2024.08.26-19.18.23:519][ 0]LogTemp: Successfully sent URI '127.0.0.1:7777' to ServerManager

```

Рисунок 12. Логи выделенного сервера после запуска

2.2. Обработка сообщений на клиенте

На клиенте была реализована удобная обработка сообщения с информацией о подключении, чтобы пользователь мог подключиться по указанному IP-адресу через графический интерфейс.

Обработка команд была реализована в отдельном потоке, так как прослушивание по сокету является блокирующей операцией. Вследствие этого, важно упомянуть, что управление виджетами средствами C++ возможно только из главного потока, и невозможно из дочернего, который был создан специально для прослушивания сообщений. Для решения такой проблемы в Unreal Engine был найден и использован метод *AsyncTask* [3], который позволяет асинхронно запустить задачу в указанном потоке (листинг 7).

Листинг 7. Работа с интерфейсом пользователя в дочернем потоке

```
const FString receivedStringData = FromBinaryArrayToString(receivedData);

GameInstance->SetConnectAddress(receivedStringData);
UMatchmakingConnectWidget* matchmakingConnectWidget = GameInstance-
>MatchMakingConnectWidget;
AsyncTask(ENamedThreads::GameThread, [matchmakingConnectWidget]()
{
    matchmakingConnectWidget->AddToViewport();
});
```

Анализируя листинг 7 можно увидеть, что указанный метод принимает первым параметром тип потока (был указан главный потока Unreal Engine), а вторым параметром была указана лямбда-функция, которая захватывает экземпляр управляемого виджета.

В итоге, когда обработка выполняется, у пользователя на экране появляется виджет, с помощью которого он может подключиться к запущенному на удаленной машине серверу (Рисунок 13).

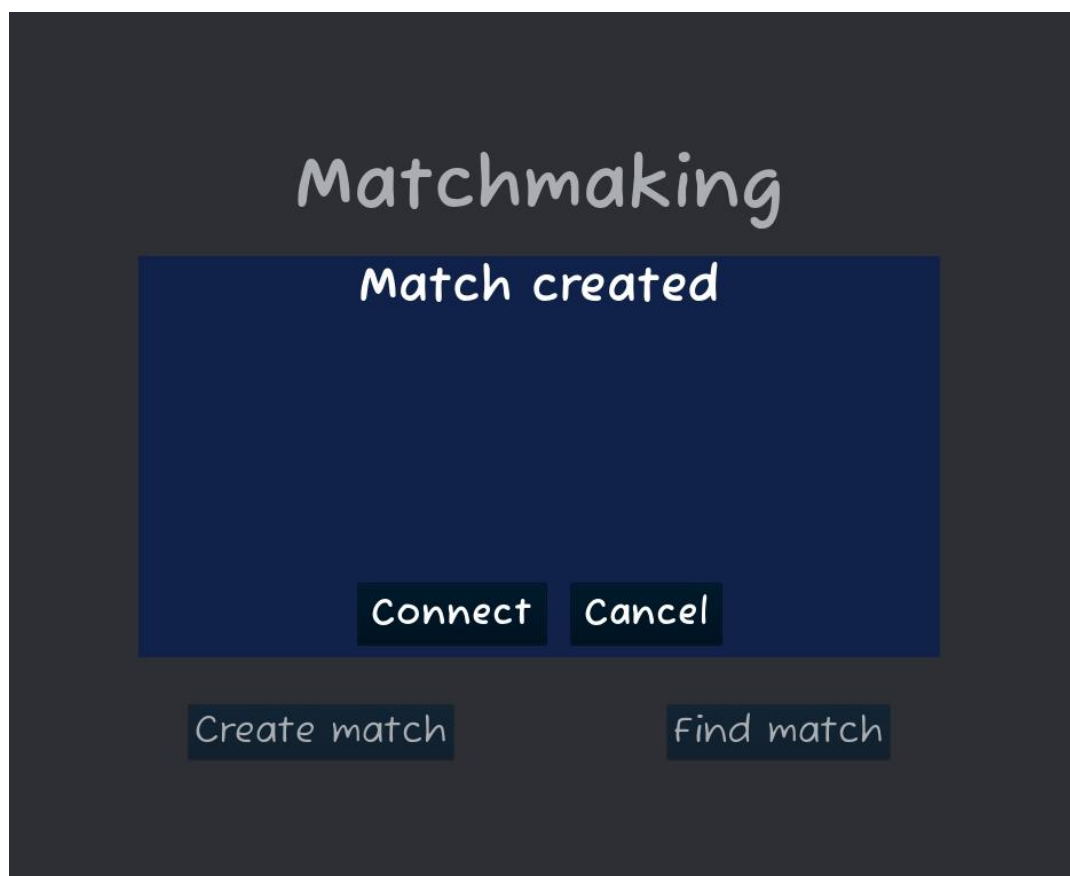


Рисунок 13. Виджет управления подключением на клиенте

2.3. Программная реализация логгера

Для того, чтобы работа системы была понятна разработчику, и чтобы работу системы можно было легче отладить, почти каждое действие, происходящее в системе, должно быть зафиксировано. Как правило, в качестве места, куда записываются логи выбирают файл [4], где каждое действие записывается в формате “<TimeStamp>: <Выполненное действие>”.

В программах менеджера серверов был реализован простой класс логгера через синглтон Майерса [5]. Синглтон Майерса позволяет через статическую переменную в классе единожды, при первой инициализации, создать экземпляр логгера и переиспользовать уже созданный экземпляр в других местах программы [6].

Пример записи событий в журнал программой с помощью созданного логгера представлен ниже (Рисунок 14).


A screenshot of a log entry displayed in a dark-themed interface. The text is white and reads: [2024-08-27 20:07:12] Start listening on 0.0.0.0:8870. The text is enclosed in a black rectangular box.

Рисунок 14. Пример записи события в журнал программы

ЗАКЛЮЧЕНИЕ

В результате выполнения практической и исследовательской работы было реализовано базовое сетевое взаимодействие клиента Unreal Engine и запущенного выделенного сервера. Для этого были выполнены задачи:

- программной реализации программы-демона;
- запуска и подготовки виртуальной среды Linux Debian 12;
- запуска программы-демона, сервера менеджеров и выделенных серверов Unreal Engine на Linux Debian;
- исследования способов обратного сетевого взаимодействия от запущенного сервера до клиента Unreal Engine;
- программной реализации доставки сообщения с запущенного выделенного сервера на клиент Unreal Engine и дальнейшей клиентской обработки данного сообщения.
- программной реализации простого логгера.

Для дальнейшей реализации и развития работы необходимо изучить и реализовать способы авторизации и аутентификации пользователя, также необходимо реализовать автоматическую систему подбора игроков с использованием уже созданной программы менеджера серверов.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Boost C++ Libraries. Gettings started on Unix Variants [Электронный ресурс] // Режим доступа: https://www.boost.org/doc/libs/1_86_0/more/getting_started/unix-variants.html (дата обращения 10.07.2024);
2. Debian. Firewall-cmd [Электронный ресурс] // Режим доступа: <https://manpages.debian.org/testing/firewalld/firewall-cmd.1.en.html> (дата обращения 12.04.2024);
3. Unreal Engine Community Wiki. AsyncTask Function [Электронный ресурс] // Режим доступа: <https://unrealcommunity.wiki/async-task-function-mfdnmfca> (дата обращения 15.04.2024);
4. Хабр. Паттерн Singleton [Электронный ресурс] // Режим доступа: <https://habr.com/ru/articles/147373/> (дата обращения 24.07.2024);
5. Хабр. Logger C++ [Электронный ресурс] // Режим доступа: <https://habr.com/ru/articles/838412/> (дата обращения 25.07.2024);
6. Хабр Q & A. Что такое пул в программировании [Электронный ресурс] // Режим доступа: <https://qna.habr.com/q/1351120> (дата обращения 26.07.2024);