|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **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 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Боженко Д.В.** |
|  | *подпись, дата* | *фамилия, и.о.* |
| Руководитель практики | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Витюков Ф. А.** |
|  | *подпись, дата* | *фамилия, и.о.* |

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Москва, 2024 г.*

**«Московский государственный технический университет имени Н.Э. Баумана**

**(национальный исследовательский университет)»**

**(МГТУ им. Н.Э. Баумана)**

УТВЕРЖДАЮ

Заведующий кафедрой \_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

« \_\_\_ » \_\_\_\_\_\_\_\_\_\_\_\_ 20 \_\_ г.

**ЗАДАНИЕ**

**на прохождение производственной практики**

эксплуатационная

тип практики

Студент

Боженко Дмитрий Владимирович \_\_\_\_\_1\_ курса группы\_\_\_РК6-21М\_\_\_\_

Фамилия Имя Отчество № курса индекс группы

в период с \_\_30\_\_. \_06\_\_\_.2024\_\_\_ г. по \_31\_\_\_. \_\_07\_\_.2024\_\_\_ г.

*Предприятие:* «НИИ Автоматизации Производственных Процессов МГТУ им. Н.Э. Баумана»

*Подразделение:* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(отдел/сектор/цех)

*Руководитель практики от предприятия (наставник):*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Витюков Федор Андреевич\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Фамилия Имя Отчество полностью, должность)

*Руководитель практики от кафедры:*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Витюков Федор Андреевич\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Фамилия Имя Отчество полностью, должность)

*Задание:*

**1.** Изучить методы запуска выделенного сервера Unreal Engine на удалённом сервере для игровой сессии средствами программы-демона на Linux Debian. Реализовать программу-демон и запустить на Linux Debian.

**2.** Исследовать варианты взаимодействия Root-менеджера с программой-демоном на удалённом сервере. Реализовать программное взаимодействие.

Дата выдачи задания « \_30\_ » \_\_июня\_\_\_\_\_\_ 2022\_ г.

Руководитель практики от предприятия  **\_\_\_\_\_\_\_\_\_\_ /\_\_** Витюков Ф.А**\_\_\_\_/**

Руководитель практики от кафедры  **\_\_\_\_\_\_\_\_\_\_ /\_\_** Витюков Ф.А.**\_\_\_/**

Студент **\_\_\_\_\_\_\_\_\_\_ /\_\_\_** Боженко Д.В. **\_\_/**

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 4](#_Toc171088695)

[реализация программы-демона 4](#_Toc171088696)

[ЗАКЛЮЧЕНИЕ 4](#_Toc171088697)

[СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ 4](#_Toc171088698)

# Ведение

# 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.

## 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 происходит 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 *сlientSocket*, 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 *сlientSocket* 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 s*tartServerInstance* 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.

## 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*.

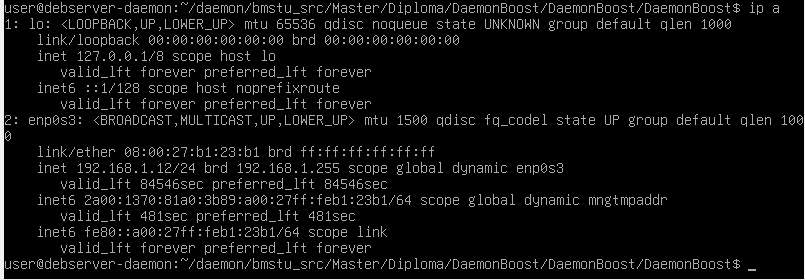


Рисунок 2. Вывод сетевых параметров ВМ

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 3. Выполнение проверки на доступность ВМ 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*

и выполнить 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>.*

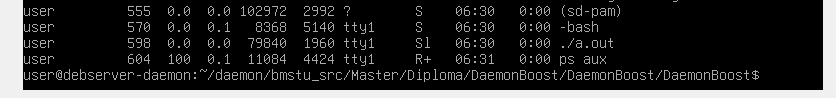


Рисунок 4. Просмотр работающего процесса через команду *ps aux*

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*. 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.



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 c 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 севера, 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 результаты обработки двух команд “Start” удаленно были запущены два экземпляра выделенных серверов, как показано на рисунке ниже.



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 c 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.

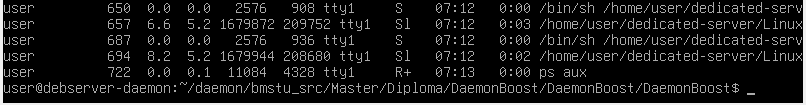


Рисунок 7. Список запущенных процессов серверов Unreal Engine

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 подключится к серверам Unreal Engine, запущенным на ВМ Linux Debian.

# Заключение

# Список используемых источников