|  |  |
| --- | --- |
| 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 (РК-6)***\_

**ОТЧЕТ ПО ПРОИЗВОДСТВЕННОЙ ПРАКТИКЕ**

Студент\_\_\_\_\_\_\_\_\_\_\_\_\_\_Боженко Дмитрий Владимирович\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*фамилия, имя, отчество*

Группа\_\_\_\_\_\_РК6-41М\_\_\_\_\_

Тип практики \_\_Преддипломная\_\_

Название предприятия \_\_\_\_\_\_\_\_\_\_\_«НИИ Автоматизации Производственных Процессов МГТУ им. Н.Э. Баумана»\_\_\_\_\_\_\_

Студент **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_Боженко Д.В.\_\_\_\_**

*подпись, дата фамилия, и.о.*

Руководитель практики

от кафедры **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_Витюков Ф.А.\_\_\_**

*подпись, дата фамилия, и.о.*

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2025 г.

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

**(национальный исследовательский университет)»**

**(МГТУ им. Н.Э. Баумана)**

УТВЕРЖДАЮ

Заведующий кафедрой *РК6*

\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_ *А.П. Карпенко* \_

« \_\_\_ » \_\_\_\_\_\_\_\_\_\_\_\_ 2025 г.

**ЗАДАНИЕ**

**на прохождение производственной практики**

**\_\_\_\_\_\_ *Преддипломная* \_\_\_\_\_\_**

Тип практики

Студент

\_\_\_\_\_\_Боженко Дмитрий Владимирович\_\_\_ \_\_ 2\_\_ курса группы \_\_\_РК6-41М\_\_

Фамилия Имя Отчество № курса индекс группы

в период с 07.02.2025 г. по 20.05.2025 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.** *Провести анализ существующих решений для реализации механизма мониторинга состояния выделенных серверов.*

**2.** *Обеспечить систему механизмом мониторинга нагрузки каждого выделенного сервера и узла системы через консольный интерфейс.*

Дата выдачи задания  *07.02.2025* г.

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

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

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

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 4](#_Toc196849183)

[1. ПОСТАНОВКА ЗАДАЧИ 5](#_Toc196849184)

[2. КРАТКИЙ ОТЧЕТ О ВЫПОЛНЕННЫХ РАБОТАХ 5](#_Toc196849185)

[3. АНАЛИЗ РЕЗУЛЬТАТОВ 11](#_Toc196849186)

[ЗАКЛЮЧЕНИЕ 13](#_Toc196849187)

[СПИСОК ЛИТЕРАТУРЫ 14](#_Toc196849188)

**ВВЕДЕНИЕ**

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.

1. **ПОСТАНОВКА ЗАДАЧИ**

Необходимо реализовать механизм мониторинга состояния выделенных серверов Unreal Engine через консольный интерфейс. Механизм мониторинга должен обновлять информацию о серверах в режиме ран-тайм и поддерживать операции сортировки и фильтрации для лучшей читаемости и восприятия данных.

1. **КРАТКИЙ ОТЧЕТ О ВЫПОЛНЕННЫХ РАБОТАХ**

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 ncurces/PDCurces [1]. Библиотека PDCurces является аналогом библиотеки ncurces, 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 ncurces/PDCurces, 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 [2] (листинг 1).

Листинг 1 Инициализация ncurces/PDCurces

void ConsoleMonitoring::Init()

{

initscr();

noecho();

cbreak();

curs\_set(0);

nodelay(stdscr, TRUE);

keypad(stdscr, TRUE);

start\_color();

init\_pair(1, COLOR\_BLACK, COLOR\_GREEN);

StartDrawLoop();

}

Метод *initscr()* инициализирует библиотеку и создает стандартный экран *stdscr* для вывод 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. *ConsoleMonitoring::StartDrawLoop()* содержит в себе логику отрисовки информации на экран, которая выполняется в бесконечном цикле (листинг 2).

Листинг 2 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 3).

Листинг 3 Функция отрисовки информации на экран

void ConsoleMonitoring::UpdateScreen()

{

clear();

DrawTableHeader();

DrawServers();

DrawFooter();

refresh();

}

Метод 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 аргументов 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 4).

Листинг 4 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 [3], *std::sort()*, принимающая в качестве параметров итераторы и лямбда-функцию. Лямбда-функция определяет правила сортировки   
(листинг 5).

Листинг 5 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::URI:

return m\_bDesendingSort ? A.m\_URI> B.m\_URI : A.m\_URI < B.m\_URI;

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:

return m\_bDesendingSort ? A.m\_serverState > B.m\_serverState : A.m\_serverState < B.m\_serverState;

default:

return true;

}

});

Для реализации направления сортировки в поля класса необходимо было добавить две переменных *bool m\_bDesendingSort* и *SortColumn::m\_sortColumn*. Переменная *m\_bDesendingSort* представляет собой булевую 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.

Как уже было упомянуто ранее, отрисовка данных в терминал является блокирующей операцией. Прослушивание входящих данных по TCP-сокету также является блокирующей операций. 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 6)

Листинг 6 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 ранее 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 7).

Листинг 7 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 программу. 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 8).

Листинг 8 Точка входа приложения менеджера серверов

int main()

{

Application application;

application.Run();

return 0;

}

1. **АНАЛИЗ РЕЗУЛЬТАТОВ**

В результате программной реализации механизма мониторинга вывод информации о запущенных выделенных серверах UE через консольный интерфейс представлен ниже (рисунок 1).



Рисунок 1 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 нажатии 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 2).



Рисунок 2 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 нижем 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.

**ЗАКЛЮЧЕНИЕ**

В результате работы была создана система мониторинга запущенных серверов Unreal Engine через интерфейс командной строки. Проведённые тестирования показали, что решение отличается высокой лёгкостью: оно минимально нагружает систему и не требует дополнительных ресурсов для работы.

Система стабильно функционирует при одновременном мониторинге множества серверов и корректно обрабатывает сетевые сбои, что подтверждает её надёжность в реальных условиях эксплуатации. Благодаря использованию CLI-подхода, мониторинг легко интегрируется в существующие инфраструктуры и подходит для автоматизации процессов.

**СПИСОК ЛИТЕРАТУРЫ**

1. ncurses(3x) – Linux manual page // man7.org [Электронный ресурс] – URL: https://man7.org/linux/man-pages/man3/ncurses.3x.html. Дата обращения 20.03.2025.
2. PDCurses Documentation // PDCurses [Электронный ресурс] – URL: https://pdcurses.org/docs/ (дата обращения: 01.04.2025).
3. C++ Standard Template Library (STL) // GeeksforGeeks   
   [Электронный ресурс] – URL: https://www.geeksforgeeks.org/the-c-standard-template-library-stl/ (дата обращения: 29.04.2025).