



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 сетевых методов автоматизированного
запуска распределенной системы выделенных серверов
Unreal Engine 4»

Студент РК6-41М
 (Группа)

Д. В. Боженко

(подпись, дата) (инициалы и фамилия)

Руководитель

Ф. А. Витюков

(подпись, дата) (инициалы и фамилия)

Москва, 2025 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

« ____ » _____ 2025 г.

ЗАДАНИЕ
на выполнение научно-исследовательской работы

по теме: Разработка сетевых методов автоматизированного запуска распределённой системы выделенных серверов Unreal Engine 4

Студент группы РК6-41М

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: 1. Исследовать принципы построения сетевых протоколов и существующие решения для обмена данными в распределённых многопользовательских системах.

2. На основе проведённого анализа спроектировать собственные структуры пакетов сетевого протокола, предназначенного для передачи данных между серверами и менеджером.

3. Реализовать распределение клиентов по запущенным экземплярам выделенных серверов.

Оформление научно-исследовательской работы:

Расчетно-пояснительная записка на 19 листах формата А4.

Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.):

Дата выдачи задания «7» февраля 2025 г.

Руководитель НИР

(Подпись, дата)

Витюков Ф.А.

И.О. Фамилия

Студент

(Подпись, дата)

Боженко Д. В.

И.О. Фамилия

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

СОДЕРЖАНИЕ

Введение	4
1. Проектирование структуры пакетов сетевого протокола	5
2. Программная реализация механизма распределения пользователей по выделенным серверам	14
Заключение	18
Список использованных источников	19

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.

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

Разработка подобной системы мониторинга и управления является важным этапом на пути к построению надёжной и масштабируемой многопользовательской инфраструктуры. Она позволит не только повысить устойчивость серверной архитектуры, но и упростит автоматизацию процессов масштабирования, обеспечивая эффективное использование ресурсов и улучшая качество взаимодействия пользователей в реальном времени.

1. ПРОЕКТИРОВАНИЕ СТРУКТУРЫ ПАКЕТОВ СЕТЕВОГО ПРОТОКОЛА

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

Обмен данными с сервером и менеджером осуществляется на основе долгоживущего TCP сокета (*long-lived-socket*). Его характерной чертой является то, что он сохраняет соединение на протяжении всей сессии, пока оно явно не разрывается одной из сторон (клиентом либо сервером). Этот подход позволяет избежать создания нового сокета для каждого сообщения, обеспечивая более эффективное взаимодействие и снижая накладные расходы на установление соединения. Для начала передачи сообщений по протоколу TCP клиентская сторона устанавливает соединение с принимающей стороной. Как только соединение было установлено сервер записывает в память сокет клиента и в бесконечном цикле пытается прочитать сообщения из клиентского сокета. Пока попытки чтения не прекращаются, клиентский сокет никогда не будет явно пытаться разорвать существующее соединение. Как только клиент присылает сообщение, сервер обрабатывает его в отдельном потоке и далее продолжает пытаться слушать следующие входящие сообщения. Если клиентский сокет явно разорвал соединение с сервером, то вызовется исключение типа *boost::system::system_error* [1], управление передается в блок *catch* и сервер явно разрывает соединение для двоих сторон выполнив функции *socket->shutdown(boost::asio::ip::tcp::socket::shutdown_both)* и *socket->close()*.

Для обработки команд, которые приходят от запущенного выделенного сервера, необходимо решить, каким образом проводить десериализацию данных.

В задачах передачи данных по сети между компонентами распределенной системы важно обеспечить корректную сериализацию и десериализацию пакетов. Существуют разные подходы реализации данной задачи. Например, (де)сериализация может производиться с использованием текстовых данных в стандартизированных форматах передачи данных как JSON или XML. Или же может быть разработан собственный протокол следующего вида: $\langle \text{имя_команды} \rangle, \langle \text{ключ}1 = \text{значение}1 \rangle, \dots, \langle \text{ключ}N = \text{значение}N \rangle$. Первым словом до разделителя, всегда идет тип команды, которую надо обработать. Далее через запятую идут пары ключ значения, которые являются опциональными и предназначены для передачи дополнительной информации к выполняемой команде. Пример команды, которая отправляется с выделенного сервера UE после его запуска для его регистрации в системе имеет вид

REGISTER_SERVER,uuid=ae34b65e4a45cd1a2,uri=127.0.0.1:7777,current_players=1,max_players=10

В команде приходят такие данные как *uuid* – уникальный идентификатор сервера, *uri* – адрес подключения, *current_players* – текущее количество игроков, *max_players* – максимально возможное количество игроков.

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

Альтернативным подходом является побайтовая (де)сериализация. Преимущества бинарной (де)сериализации включают в себя высокую скорость (де)сериализации, малый размер передаваемых пакетов за счет ручного управления размеров каждого поля и строгую структурированность данных. К недостаткам бинарной сериализации и десериализации, несомненно, относятся трудности при отладке и проблемы совместимости между разными

архитектурами из-за разной размерности типов данных и разного порядка байтов. Однако описанную проблему необходимо и возможно разрешить. Следовательно, для принятия решения о используемом подходе передачи данных необходимо было провести исследование, провести анализ возможного использования каждого из подходов и выполнить замеры производительности.

Для строковой сериализации данных достаточно лишь сформировать строку, переконвертировать полученную строку в массив байтов и отправить полученную полезную нагрузку по сокету (листинг 1). Клиент, в данном случае выделенный сервер UE, не должен беспокоиться о порядке передачи байтов, так как все заложено в строке.

Листинг 1 — Метод отправки строковых данных в менеджер

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

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

Для сериализации данных на стороне менеджера (листинг 2) достаточно просто записать полученные из сокета байты в строку и далее производить ее парсинг, например, по символу разделителю '='.

Листинг 2 — Десериализация строковых данных на стороне менеджера

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

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

Также помимо регистрации сервера необходимо было обработать событие, когда меняется текущее количество игроков. В классе *AGameMode* есть виртуальный метод *virtual void PostLogin(APlayerController* NewPlayer)*, который вызывается каждый раз как к серверу подключается новый пользователь. Аналогичную логику необходимо было реализовать и тогда, когда пользователь отключается от сервера. Для реализации данной задачи был переопределен и использован метод *virtual void Logout(AController* Exiting)*, который выполняется каждый раз, как пользователь отключается от сервера. Пример логов менеджера, когда сервер был запущен и к нему подключилось два пользователя представлен ниже (рисунок 1).

```

3 [2025-03-18 23:55:55.893] Starting applicaton...
4 [2025-03-18 23:55:55.903] Start listening requests on 0.0.0.0:8870
5 [2025-03-18 23:57:03.406] Got data from client: CREATE
6 [2025-03-18 23:57:03.410] Sending "START" command to daemon due to empty running server instances array
7 [2025-03-18 23:57:03.412] Sending command to daemon: START
8 [2025-03-18 23:57:03.418] Added client to queue: [CLIENT_ADDRESS=127.0.0.1:55591, CLIENT_TYPE=INITIATOR]
9 [2025-03-18 23:57:14.754] Getting data from dedicated server: REGISTER_SERVER,uuid=1D6B4B5D4194BE7B571B3
10 [2025-03-18 23:57:14.759] Registering server job started...
11 [2025-03-18 23:57:14.766] Registered server with uuid = 1D6B4B5D4194BE7B571B349038436F24
12 [2025-03-18 23:57:14.767] Sending uri of started dedicated server to client: 127.0.0.1:7777
13 [2025-03-18 23:57:14.768] Senging data to cleint: 127.0.0.1:7777
14 [2025-03-18 23:57:14.770] Registering server job finished...
15 [2025-03-18 23:57:15.376] Getting data from dedicated server: UPDATE_SERVER,uuid=1D6B4B5D4194BE7B571B349
16 [2025-03-18 23:57:15.379] Updating server job started...
17 [2025-03-18 23:57:15.381] Old value: [current_players = 0, state = LOBBY]
18 [2025-03-18 23:57:15.383] New value: [current_players = 1, state = LOBBY]
19 [2025-03-18 23:57:15.384] Updating server job finished...
20 [2025-03-18 23:57:45.270] Got data from client: CREATE
21 [2025-03-18 23:57:45.272] Found server instance [uuid=1D6B4B5D4194BE7B571B349038436F24] for client [CLIE
22 [2025-03-18 23:57:45.274] Senging data: [URI=127.0.0.1:7777] to cleint: [CLIENT_ADDRESS=127.0.0.1:55624,
23 [2025-03-18 23:57:46.084] Getting data from dedicated server: UPDATE_SERVER,uuid=1D6B4B5D4194BE7B571B349

```

Рисунок 1 — Лог менеджера при подключении пользователей
(строковая десериализация)

На рисунке видно, что при одном из запусков при строковой десериализации с учетом ввода вывода данных в поток процесс регистрации сервера занимает 12 мс и процесс обновления информации о сервере занимает 8 мс. Данное значение невелико, однако, стоит его сравнить с байтовой реализацией.

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

который необходимо (де)сериализовать и передавать по сети представлен ниже (листинг 3).

Листинг 3 – Структура данных команды регистрации сервера

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

На листинге видно, что используется такие типы данных как *uint16_t*, *uint8_t*. Данные типы данных находятся в заголовочном файле библиотеки *cstdint*, и они предназначены для чего, чтобы четко указать размер поля в структуре. К примеру, поле *m_serverState* – это небольшой *enum*, который содержит в себе перечисление возможных состояний запущенного сервера UE. Данное количество строго ограничено, поэтому для его хранения будет достаточно одного байта. Количество подключенных пользователей – это не строго ограниченное по размерам поле, оно может принимать разные значения. Следовательно, размер поля в два байта обеспечивает достаточный диапазон значений (0 - 65536) для правильного хранения. Поле *m_uuid* представляет собой UUID сервера, который имеет всегда одинаковую длину в 36 символов. Такой идентификатор можно хранить в виде строки, что будет занимать 37 символов с учетом терминального символа строки ‘\0’. Гораздо эффективней будет хранить данное поле в виде 16 байт, так как для представления одного числа в шестнадцатеричной системе счисления необходимо 4 бита. Также можно поступить и с передачей адреса и порта. Гораздо эффективней хранить IP-адрес в виде 4 байт, а порт – в размере 2 байт, так как максимального значения в 65 536 будет вполне хватать для указания порта, на котором запустился сервер. *#pragma pack (push, 1)* и *#pragma pack (pop)* являются директивой препроцессора и служат для того, чтобы указать компилятору на необходимость отключения байтового выравнивания на данном участке кода.

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

Для сериализации данных на отправляющей стороне достаточно выделить память под нужную структуру данных, заполнить ее полезной нагрузкой и записать в массив байт, который потом передается по сокету (листинг 4).

Листинг 4 – Байтовая сериализация на стороне сервера UE

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

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

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

MessageFrameHeader frameHeader;

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

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

В листинге выше видно, что помимо полезной нагрузки в сокет также отправляется структура данных *MessageFrameHeader* (метаинформация). Данная структура состоит из поля *uint8_t CommandType*, обозначающим тип отправляемой команды весом в один байт; и поля *uint16_t PayloadSize*, обозначающим размер полезной нагрузки в байтах, которая идет следом за

метаинформацией. В итоге ожидаемый размер структуры для команды регистрации представлен ниже (рисунок 2).

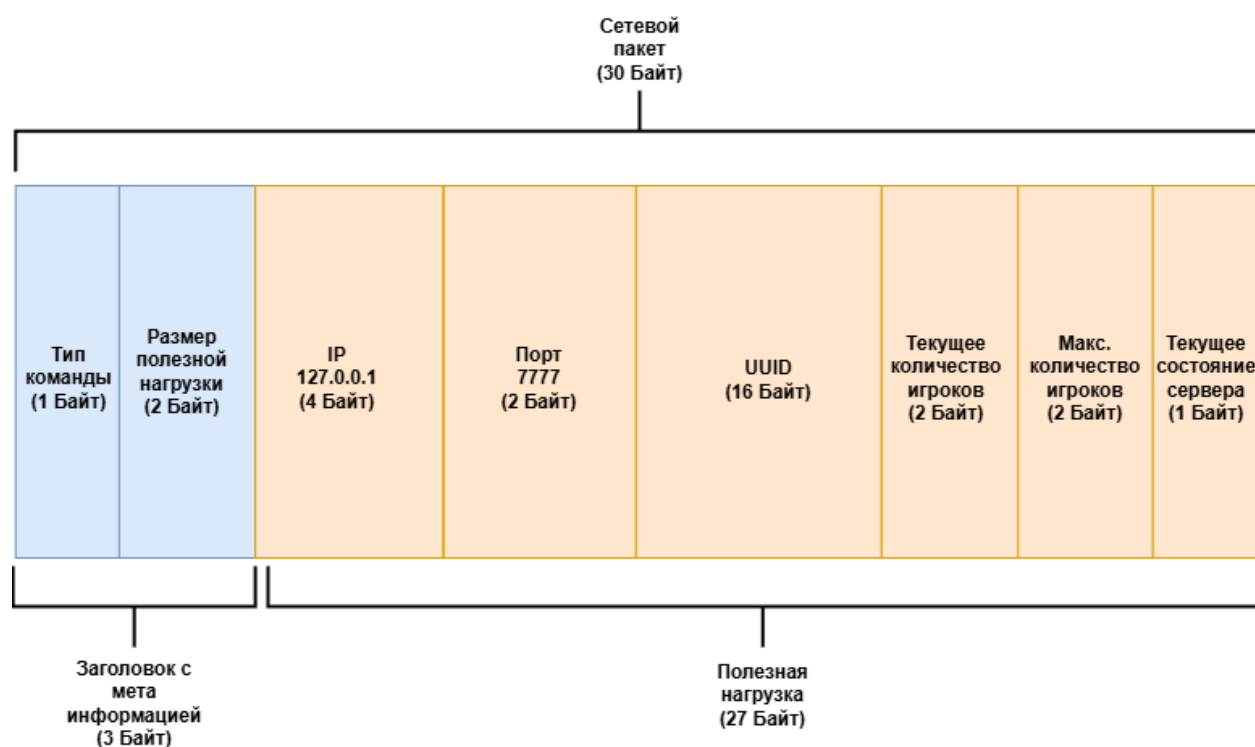


Рисунок 2 – Побайтовая схема сообщения регистрации сервера

С использованием такой структуры байтов в сетевом пакете при получении данных из сокета принимающая сторона понимает, какая команда пришла, какой размер полезной нагрузки, сколько памяти надо выделить под полезную нагрузку, что в дальнейшем ее десериализовать (листинг 5).

Листинг 5 – Десериализация данных на стороне менеджера

```
try
{
    while (true)
    {
        MessageFrameHeader frameHeader;
        boost::asio::read(*socket, boost::asio::buffer(&frameHeader,
sizeof(MessageFrameHeader)));

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

        ProcessBinaryDataFromServer(frameHeader, payload);
    }
}
```

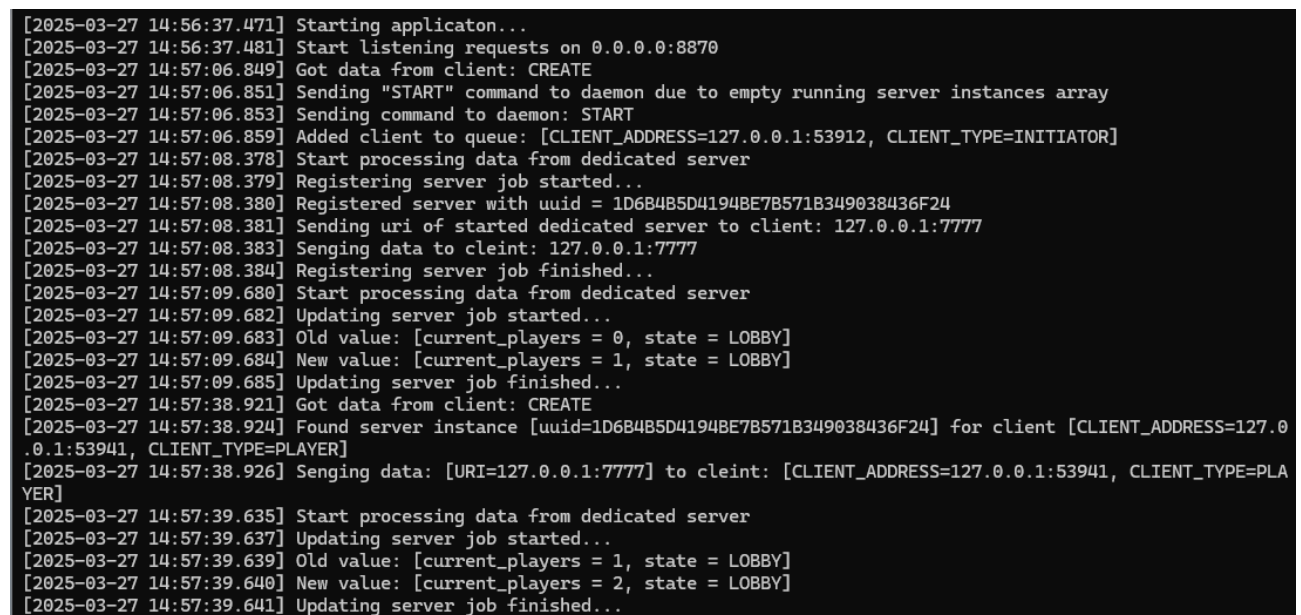
При десериализации полезной нагрузки и разбиении ее на поля необходимо обязательно проверить, совпадает ли размер полученной нагрузки с ожидаемой. Если размер совпадает, то можно предпринять попытку копирования массива байтов в структуру данных, иначе – необходимо пропустить полученную команду и вывести соответствующее сообщение (листинг 6).

Листинг 6 – Обработка массива байт полезной нагрузки на стороне менеджера

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

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

Ниже на рисунке представлен лог менеджера при реализации байтовой (де)сериализации (рисунок 3).



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

Рисунок 3 – Лог менеджера при подключении пользователей
(байтовая десериализация)

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

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

2. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ МЕХАНИЗМА РАСПРЕДЕЛЕНИЯ ПОЛЬЗОВАТЕЛЕЙ ПО ВЫДЕЛЕННЫМ СЕРВЕРАМ

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

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

Если работающие серверы уже существуют, осуществляется фильтрация тех из них, которые имеют свободные места для новых игроков, но при этом уже содержат хотя бы одного подключённого пользователя. Это позволяет минимизировать количество частично заполненных серверов и более эффективно использовать ресурсы.

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

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

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

В многопоточных сетевых приложениях, таких как сервер на основе сетевой библиотеки `boost::asio` [2], одновременная обработка данных от

множества клиентов является нормальной рабочей ситуацией. Каждый новый клиентский запрос может обрабатываться в отдельном потоке или через пул потоков, чтобы повысить производительность системы. В условиях многопоточности возникает одна из наиболее серьезных проблем – гонка данных. Она проявляется тогда, когда два или более потока одновременно обращаются к одной и той же области памяти, при этом хотя бы один из них выполняет запись. Без надлежащей синхронизации это приводит к непредсказуемому поведению программы: повреждению данных, крахам, утечкам памяти или логическим ошибкам. Для устранения такой проблемы был применен механизм синхронизации `std::mutex` (листинг 7).

Листинг 7 – Применение механизма синхронии при обработке запроса клиентов

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

    ClientInfo clientInfo;
    clientInfo.Socket = socket;

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

    if (_runningServers.empty())
    {
        ...
        return;
    }
    ...
    SendDataToSocket(socket, runningServerWithMostPlayers.m_URI);
}
```

В рассматриваемом листинге метод *ProcessDataFromClient* обращается к контейнеру *_runningServers*, который хранит информацию о запущенных выделенных серверах. Поскольку одновременно несколько клиентов могут инициировать обработку своих запросов, возникает необходимость синхронизировать доступ к этому ресурсу. Например, в момент, когда один поток перебирает список серверов для поиска подходящего инстанса, другой поток мог бы изменить структуру этого списка, добавив, изменив или удалив сервер (листинг 8).

Листинг 8 – Применение механизма синхронии при обработке запроса серверов

```
void TcpServer::ProcessBinaryDataFromServer(const MessageFrameHeader& header,
const char* payload, const size_t payloadSize)
{
    ServerCommandType commandType =
static_cast<ServerCommandType>(header.m_commandType);

    switch (commandType)
    {
        case ServerCommandType::REGISTER_SERVER:
        {
            Logger::GetInstance() << "Registering server job started..." <<
std::endl;
            std::lock_guard<std::mutex> lock(_runningServersMutex);
            ...
            break;
        }
        case ServerCommandType::UPDATE_SERVER:
        {
            Logger::GetInstance() << "Updating server job started..." <<
std::endl;
            std::lock_guard<std::mutex> lock(_runningServersMutex);
            ...
            break;
        }
        default:
            Logger::GetInstance() << "Unknown command type from server, skip
processing data..." << std::endl;
    }
}
```

Для предотвращения подобных ситуаций используется механизм взаимного исключения. В методах дополнительно применяется обёртка `std::lock_guard<std::mutex>`, которая гарантирует, что доступ к `_runningServers` возможен только одному потоку за раз. Благодаря этому достигается целостность данных и исключается риск гонок. Применение *lock_guard* [3] делает код более безопасным, так как мьютекс автоматически освобождается при выходе объекта за пределы области видимости. Блок-схема алгоритма распределения пользователей по запущенным серверам представлена ниже (рисунок 4).

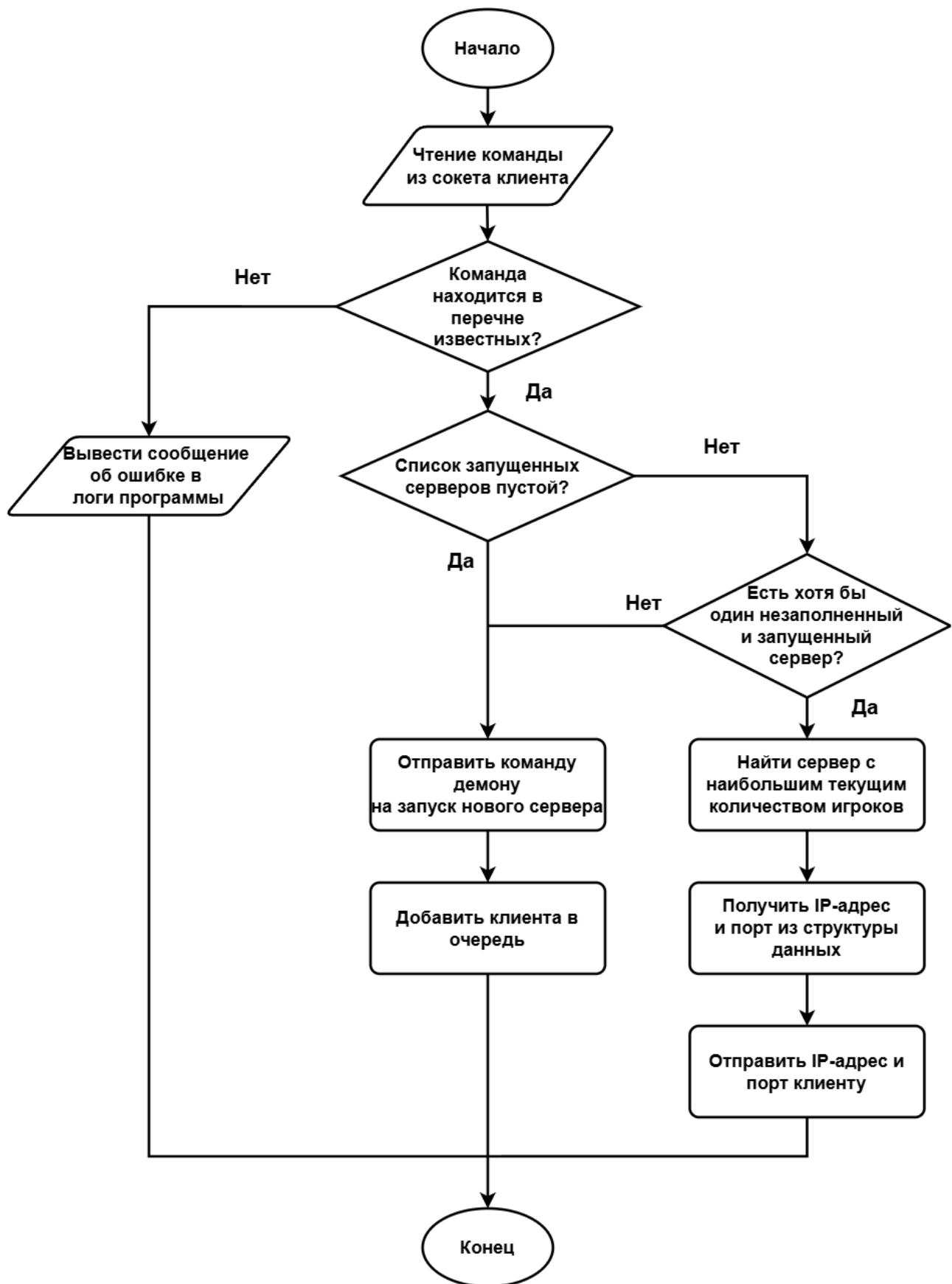


Рисунок 4 – Блок-схема алгоритма распределения пользователей по выделенным серверам

ЗАКЛЮЧЕНИЕ

В ходе работы была проведена исследовательская и практическая деятельность, направленная на проектирование и реализацию механизма взаимодействия между клиентами и выделенными серверами Unreal Engine. В ходе исследования были изучены подходы к проектированию сетевых протоколов, после чего спроектирована собственная структура сетевого пакета, учитывающая требования к эффективности, читаемости и расширяемости.

Особое внимание было уделено сравнению строковой и бинарной сериализации: в результате анализа выявлено, что бинарная форма передачи данных обеспечивает меньший объём пакетов и более высокую производительность при передаче, что особенно важно в условиях сетевой нагрузки и ограниченной пропускной способности. На основе выбранного формата была реализована система отправки и обработки команд, каждая из которых представляет собой сериализованный сетевой пакет.

Реализация данного механизма позволила организовать распределение пользователей по выделенным серверам в зависимости от текущей загрузки и состояния серверной инфраструктуры.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Boost.Asio // Boost C++ Libraries [Электронный ресурс] – URL: https://www.boost.org/doc/libs/1_76_0/doc/html/boost_asio.html (дата обращения 23.03.2025).
2. Boost.Asio C++ Network Programming // Habr [Электронный ресурс] – URL: <https://habr.com/ru/articles/195794/> (дата обращения 01.04.2025).
3. std::lock_guard // cppreference.com [Электронный ресурс] – URL: https://en.cppreference.com/w/cpp/thread/lock_guard/ (дата обращения 20.04.2025).