Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ОРЕНБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ В. А. БОНДАРЕНКО»

Институт математики и информационных технологий

Кафедра математики и цифровых технологий

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Направление подготовки 02.03.02 Фундаментальная информатика и информационные технологии

Разработка менеджера сообщений для игры Minecraft

ОГУ 02.03.02. 1326. 440 ОО

Заведующий кафедрой
канд. пед. наук, доцент А. Е. Шухман

Руководитель
канд. пед. наук, доцент А. Е. Шухман

Студент А. С. Мочалин

Оренбург 2026

Утверждаю
заведующий кафедрой математики
и цифровых технологий
_____А. Е. Шухман
«____»_____20__ г.

ЗАДАНИЕ

на выполнение выпускной квалификационной работы

студенту Мочалину Артёму Сергеевичу

по направлению подготовки 02.03.02 Фундаментальная информатика и информационные технологии

1 Тема ВКР: Разработка менеджера сообщений для игры Minecraft
2 Срок сдачи студентом ВКР: «___»_____2026 г.
3 Цель и задачи ВКР:
Цель – разработать менеджер сообщений для игры Minecraft, обеспечивающий маршрутизацию по каналам, фильтрацию, модерацию и синхронизацию сообщений в распределенной сети.
Задачи:
– изучить теоретические аспекты разработки системы;
– описать функциональные требования к разрабатываемой системе;
– спроектировать модель системы с использованием современных средств и методов;
– описать алгоритмы и архитектурные особенности разрабатываемой системы;
– реализовать систему и описать этапы разработки;
– протестировать реализованную систему.
4 Исходные данные к ВКР: научные публикации и литература, содержащие информацию об асинхронной обработке сетевых пакетов, системах модерации сообщений и распределенных серверных архитектурах.
5 Перечень вопросов, подлежащих разработке: осуществить обзор литературы по теме исследования; проанализировать существующие аналоги разрабатываемой системы; обосновать выбор инструментальных средств; описать функциональные требования; спроектировать модель системы; описать алгоритмы и архитектурные особенности разрабатываемой системы; реализовать и протестировать систему; апробировать полученные результаты.
6 Перечень графического (иллюстративного) материала: рисунки и таблицы, необходимые для изложения материала.

Дата выдачи и получения задания

Руководитель «___»_____20__ г. А. Е. Шухман

Студент «___»_____20__ г. А. С. Мочалин

Аннотация

Выпускная квалификационная работа посвящена разработке менеджера сообщений для серверов игры Minecraft, обеспечивающего маршрутизацию сообщений по каналам, модерацию контента и кросс-серверную синхронизацию. В работе рассматриваются преимущества модульной архитектуры, способы низкоуровневого перехвата сетевых пакетов, организация конвейерной обработки сообщений и асинхронного взаимодействия между серверами.

Работа состоит из введения, трех глав, заключения, списка использованных источников и одного приложения.

Первая глава посвящена теоретическим аспектам разработки системы. В ней проведен анализ зарубежных и отечественных источников по обработке сетевого трафика в Minecraft и построению высоконагруженных серверных систем, а также выполнен сравнительный анализ существующих систем управления сообщениями.

Вторая глава посвящена проектированию системы. В ней сформулированы функциональные требования к разрабатываемому приложению, обоснован выбор инструментальных средств разработки, разработана структурная модель системы с описанием состава модулей и принципов их взаимодействия.

Третья глава посвящена реализации и тестированию системы. Описаны архитектурные особенности, приведены алгоритмы обработки сообщений и инициализации, а также результаты функционального и стресс-тестирования, подтвердившие стабильную работу при высокой нагрузке.

В приложении А представлены сертификаты и благодарность, подтверждающие участие в научных конференциях.

Работа выполнена на 41 странице с использованием 20 источников и содержит 11 рисунков, 3 таблицы и приложение.

Содержание

Введение

В мире видеоигр Minecraft занимает особое место. Это большая платформа для творчества и социального взаимодействия, где миллионы игроков строят миры, создают проекты и общаются друг с другом. Основным местом для этого общения служит внутриигровой чат.

Со временем стандартный чат перестает справляться. Все игроки пишут сообщения в один поток, а важные объявления от сервера быстро теряются и администраторам трудно выявлять нарушения. Эту задачу решает менеджер сообщений. Он разбивает разные сообщения в разные каналы, например торговля или общий разговор, и каждый игрок видит только то, что он хочет, не отвлекаясь на все сообщения, а администраторам легче выявлять нарушения через автоматическую проверку.

Сервер может быть более сложным и состоять из двух или более backend-серверов, которые связаны между собой через proxy-сети. Minecraft не имеет инструмента, чтобы связывать сообщения между этими серверами. Эту проблему решает менеджер, который синхронизирует сообщения через proxy с помощью серверной базы данных, а игроки могут общаться в одном канале, не замечая разницы между серверами.

Не менее важна производительность. Если у сообщений будет задержка после того, как игрок его написал, это будет портить игровой опыт. Поэтому система должна быть асинхронной, где работа над сообщениями происходит в разных потоках. Один поток проверяет поступивший текст на запрещенные слова, второй форматирует следующее сообщение, а третий отправляет прошлое сообщение получателям. Такой конвейер не блокирует игровой процесс и минимально задерживает отправку сообщений при большом количестве игроков, что важно для крупных серверов.

Уже существуют готовые решения для чата, например EssentialsX Chat, Chatty, VentureChat и другие. Однако их возможности ограничены для крупного проекта, либо избыточны для небольшого сообщества. Их трудно настроить под конкретные задачи без изменения исходного кода. Они привязаны к определенной платформе Bukkit или Fabric и не дают владельцу сервера возможности экспериментировать с разными настройками без потери функционала.

Тема исследования актуальна потому, что владельцы серверов сталкиваются с ограничениями готовых решений. Встроенная реализация Minecraft чата ограничена, а многие плагины, которые должны решать эту проблему, замедляют работу системы и не имеют полноценной модульной архитектуры. Такой менеджер станет не просто утилитой, а важной частью инфраструктуры, напрямую влияющей на комфорт игроков и эффективность работы администрации.

Целью выпускной квалификационной работы является разработка менеджера сообщений для игры Minecraft.

Для достижения поставленной цели необходимо решить следующие задачи:

1) изучить теоретические аспекты разработки системы;
2) описать функциональные требования к разрабатываемой системе;
3) спроектировать модель системы с использованием современных средств и методов;
4) описать алгоритмы и архитектурные особенности разрабатываемой системы;
5) реализовать систему и описать этапы разработки;
6) протестировать реализованную систему.

1 Теоретические аспекты разработки системы

1.1 Анализ зарубежных и отечественных источников

В ходе выпускной квалификационной работы по разработке менеджера сообщений для игры Minecraft были рассмотрены и изучены зарубежные исследования в области разработки игровых систем, обработки сообщений и высоконагруженных приложений. Список этих исследований и их краткое содержание представлены в таблице 1.

Таблица 1 – Исследования зарубежных авторов в области управления сообщениями для игры Minecraft

Зарубежные исследования		
Наименование работы	Автор	Краткое содержание
Distributed architecture for online multiplayer games (Распределенная архитектура для многопользовательских онлайн-игр) [9]	A. Bharambe	В статье предложена распределенная архитектура для многопользовательских игр. Система позволяет разбивать нагрузку между несколькими серверами, сохраняя низкие задержки и согласованность игрового мира при большом числе игроков.
Large scale in-game spectating system for Minecraft-like games (Крупномасштабная внутриигровая система наблюдения за Майнкрафтом) [10]	M. Delgorge	В документе описывается система трансляции состояния сервера Minecraft внешним зрителям. В такой пакетной модели все обновления мира попадают в очередь, после чего компонент обработки сообщений фильтрует события и перенаправляет их для обслуживания зрителей.
Distributed Server for the Game (Распределенный сервер для игры) [13]	T. Hejman	В работе рассматривается использование функционального программирования для реализации параллельного сервера Minecraft. Автор строит собственную версию сервера на Scala, показывая преимущество модели.
Transforming Minecraft into a research platform (Превращение Minecraft в исследовательскую платформу) [14]	A. E. Herman	В статье предлагается архитектура, модифицирующая стандартный сервер Minecraft. В нее введен прокси и P2P-слой, что позволяет перераспределять обработку зон мира между узлами.

Также были рассмотрены отечественные исследования в области управления сообщениями для игры Minecraft. Список этих исследований и их краткое содержание представлены в таблице 2.

Таблица 2 – Отечественные исследования в области управления сообщениями для игры Minecraft

Отечественные исследования		
Наименование работы	Автор	Краткое содержание
Разработка высоконагруженного игрового WebSocket-сервера на Java [1]	А. А. Балцер	В статье рассматривается реализация масштабируемого WebSocket-сервера для браузерной онлайн-игры на Java и Netty.
Разработка сессионного контейнера серверного Java-кода [2]	В. С. Виноградов	В статье приводится концепция и реализация контейнера серверного кода на Java, обеспечивающего постоянное клиент-серверное соединение.
Высокопроизводительный сервер на Netty [3]	Eirenliel	В статье рассматриваются проблемы масштабируемости Minecraft-серверов при использовании традиционной модели с использованием одного потока на соединение.

Таким образом, были изучены отечественные и зарубежные литературные источники. Для дальнейшей работы были отобраны:

1) исследование «Large scale in-game spectating system for Minecraft» (M. Delgorge), которое предлагает практические решения по фильтрации и маршрутизации игровых сообщений;

2) работа «Distributed Server for the Game Minecraft» (T. Hejman), в которой представлены подходы к модульности на основе функциональной декомпозиции, позволяющие создавать тестируемые и поддерживаемые компоненты обработки сетевых пакетов;

3) статья «Разработка сессионного контейнера серверного Java-кода» (В. С. Виноградов), демонстрирующая методы организации долгоживущих соединений;

4) исследование «Высокопроизводительный NIO-сервер на Netty» (Eirenliel), которое описывает методы оптимизации производительности, включая асинхронную обработку соединений и эффективное управление буферами.

Указанные работы в совокупности предоставляют необходимый теоретический и практический фундамент для создания менеджера сообщений, сочетающего гибкость архитектуры с эффективностью обработки данных.

1.2 Анализ существующих аналогов разрабатываемой системы

Современные многопользовательские серверы в Minecraft требуют управления сообщениями между игроками. Плохо организованный чат может привести к спаму, использованию ненормативной лексики и в целом ухудшить игровой опыт, что негативно сказывается на росте и удержании игровой аудитории. Для решения этих задач используются специализированные программные расширения (плагины и моды), которые предоставляют администраторам инструменты для форматирования, модерации и настройки внутриигрового чата. Далее рассмотрим наиболее популярные и функциональные решения в этой области.

EssentialsX Chat является модулем в составе плагина EssentialsX, который представляет собой набор основных конфигураций для серверов, работающих на платформах Bukkit, Spigot, Paper и их производных. Его ключевые особенности включают настраиваемые форматы сообщений с поддержкой цветов и подстановочных переменных, а также возможность создания отдельных каналов для различных групп игроков, определяемых системами разрешений. Главным достоинством является тесная интеграция с другими модулями EssentialsX и высокая стабильность. К основным недостаткам можно отнести ограниченные возможности для полной кастомизации.

Chatty представляет собой самостоятельный плагин, который имеет расширенные возможности настройки чата. Он позволяет изменять шрифты и цвета сообщений, а также поддерживает приватное общение между игроками. Основной особенностью Chatty является настройка сообщений под себя. Но чтобы использовать плагин, администратору приходится изучать сложную структуру конфигурации.

VentureChat поддерживает работу в распределенных сетях серверов через BungeeCord и Velocity. Можно настроить многоуровневую систему каналов с проверкой прав доступа и собственными командами. Плагин поддерживает интеграцию с системами разрешений LuckPerms и синхронизирует чат между серверами в proxy сети. Основным недостатком является отсутствие средств модерации сообщений.

DiscordSRV интегрируется с мессенджером Discord. Он отправляет сообщения между текстовыми каналами Discord и внутриигровым чатом Minecraft, что дает игрокам возможность общаться за пределами игры. Однако плагин не форматирует обычные сообщения и не имеет инструментов модерации, поэтому он используется с другими решениями вместе, а не самостоятельно.

BetterChat работает на платформе Fabric. Он добавляет упоминания игроков через специальный символ, который настраивается. Также имеет историю сообщений и фильтрует нарушения игроков. Он создан только для Fabric, поэтому его нельзя использовать с другими решениями вместе на Bukkit.

DeluxeChat настраивает сообщения под конкретного игрока сервера. Администратор может выдавать игрокам префиксы или суффиксы и ставить

разные форматы чата. Главным преимуществом является простота настройки. В свою очередь, функционал, связанный с модерацией, в нем развит слабее, чем в специализированных аналогах.

InteractiveChat – плагин, предоставляющий расширенные возможности форматирования текста. Он позволяет использовать элементы интерактивности в сообщениях, например всплывающие подсказки, текст при наведении курсора, кликабельные команды. Недостатком является необходимость установки вручную дополнительных библиотек на сервер.

Изучив каждое решение, проведем сравнительный анализ данных по нескольким критериям и результат отразим в таблице 3.

Таблица 3 – Сравнительный анализ имеющихся решений в области управления сообщениями для игры Minecraft

Критерий сравнения	Наименование продукта						
	EssentialsX Chat	Chatty	Venture Chat	Discord SRV	Better Chat	Deluxe Chat	Interactive Chat
Форматирование сообщений	–	+	+	+	+	+	+
Система каналов	+	+	+	–	–	–	+
Кросс-серверность	–	–	+	+	–	–	+
Интеграция с Discord	–	–	–	+	–	–	–
Инструменты модерации	+	–	–	–	+	–	–
Интерактивные элементы	–	–	–	–	–	–	+
Простота настройки	+	–	–	+	+	+	–

По результатам анализа можно сделать вывод о том, что системы управления демонстрируют различные подходы к организации внутриигровых сообщений, причем решения EssentialsX Chat и BetterChat показали ограниченную функциональность по сравнению с VentureChat и InteractiveChat. Ни одна из существующих систем не предоставляет комплексного решения, сочетающего развитую систему, инструменты модерации и расширенные возможности визуальной кастомизации. Наиболее перспективными для использования являются VentureChat с его кросс-серверной синхронизацией и InteractiveChat с интерактивными элементами форматирования.

При проектировании системы были учтены выявленные ограничения существующих решений, а также реализованы ключевые функции успешных аналогов с модульной архитектурой, поддержкой распределенных сетей серверов, гибкой системой прав доступа и расширенными возможностями форматирования сообщений.

2 Проектирование системы с использованием современных средств и методов

2.1 Функциональные требования к разрабатываемой системе

Целью разрабатываемого менеджера сообщений является преобразование стандартных сообщений Minecraft в управляемую систему с возможностями модерации и кастомизации. Разрабатываемая система должна обеспечивать стабильную работу при нагрузке, поддерживать настройку под конкретный сервер и интегрироваться с существующей инфраструктурой Minecraft-серверов.

Для достижения поставленной цели система должна обеспечивать следующие функциональные требования:

1) перехват и обработка сетевых пакетов чата на низком уровне;
2) организация системы сообщений с разделением прав доступа;
3) применение цепочки фильтров для модерации содержимого сообщений;
4) динамическое форматирование сообщений;
5) обеспечение синхронизации в распределенных сетях.

Чтобы эффективно обеспечивать поставленные требования, система разделена на несколько взаимосвязанных модулей, каждый из которых отвечает за определенную функциональную область. Такое разделение позволяет упростить разработку и обеспечивает возможность независимого обновления отдельных компонентов. Модульная архитектура дает возможность администраторам серверов отключать неиспользуемые функции и расширять систему через API. Ключевыми модулями системы являются:

1. Модуль перехвата пакетов, который работает на базе библиотеки PacketEvents [19]. Этот модуль перехватывает сетевые пакеты чата на уровне Netty, обеспечивая асинхронную обработку без блокировки основного потока. Модуль получает сырые данные сообщений и передает их в ядро системы для дальнейшей обработки, а также отправляет готовые сообщения обратно игрокам после прохождения всей цепочки модерации и форматирования.

2. Модуль управления каналами. Он позволяет создавать различные типы каналов. Каждый канал имеет собственные настройки видимости, форматы сообщений и прав доступа. Модуль определяет, в какой канал направляется сообщение игрока, и какие игроки должны его увидеть. Игрок может находиться сразу в нескольких каналах и для каждого изменять собственные уведомления.

3. Модуль конвейера обработчиков. Сообщение проходит через цепочку обработчиков, которая работает по принципу конвейера. Сюда входят обработчик форматирования, обработчик подстановки переменных и другие компоненты. Каждый обработчик выполняет только свою задачу, после чего

передает результат дальше. Если на каком-то этапе сообщение отклоняется, то вся цепочка прерывается, а отправитель получает сообщение об ошибке.

4. Модуль модерации. Он выявляет нарушения по заданным фильтрам. Каждое такое нарушение записывается в базу данных. Применяются проверки по черным и белым спискам, по частоте отправки сообщений и по количеству букв в тексте.

5. Модуль работы с данными. Он использует абстрактный слой доступа к данным, который поддерживает работу с различными СУБД: H2, SQLite, MySQL, MariaDB и PostgreSQL. Модуль отвечает за сохранение настроек чата, истории модерации и данных игроков.

6. Модуль кросс-серверной синхронизации. Он синхронизирует состояние чата между всеми серверами сети, позволяя игрокам общаться независимо от того, на каком конкретном сервере они находятся в данный момент.

Таким образом, сформулированные функциональные требования и выделенные модули образуют архитектуру разрабатываемого менеджера сообщений, обеспечивающую выполнение задач по перехвату, модерации, форматированию и кросс-серверной синхронизации сообщений.

2.2 Выбор инструментальных средств для разработки

При выборе средств для разработки менеджера сообщений учитывались специфические требования, связанные с особенностями работы в среде Minecraft, необходимостью обработки большого объема сообщений в реальном времени, а также обеспечением совместимости с существующей инфраструктурой игровых серверов.

Основным языком программирования для реализации системы был выбран язык Java, потому что официальная серверная реализация Minecraft полностью написана на Java. Стандартная библиотека Java имеет встроенные средства многопоточного программирования в пакете java.util.concurrent, например CompletableFuture, AtomicBoolean, CopyOnWriteArrayList и другие [16]. Это важно для асинхронной обработки сообщений, которые поступают одновременно от игроков. Строгая статическая типизация языка помогает выявлять ошибки на этапе компиляции. Экосистема Java имеет множество библиотек и фреймворков и не зависит от операционной системы.

Для управления зависимостями и сборки используется Gradle Groovy [12]. В отличие от Maven, скрипты на Groovy пишутся проще и быстрее, чем XML-конфигурация, и имеется встроенное кэширование, что сокращает повторную сборку проекта. В конфигурации сборки созданы профили для серверных платформ Bukkit, Paper, Fabric, BungeeCord и Velocity, автоматически разрешающие транзитивные зависимости между библиотеками.

Используется внедрение зависимостей с помощью библиотеки Google Guice [8]. Компоненты ссылаются на классы, а не на реализации, что ослабляет

связи внутри системы. Аннотация @Inject описывает зависимости между модулями, а код становится проще для чтения и поддержки. Модули можно заменять в зависимости от платформы, что позволяет создавать адаптеры к конкретной реализации, а логику системы не изменять.

Для доступа к базе данных используется абстрактный слой, который поддерживает различные СУБД, включая H2, SQLite, MySQL, MariaDB, PostgreSQL. Это дает возможность администратору сервера выбирать подходящую базу данных. H2 и SQLite подходят для небольших серверов, потому что не требуют настройки [4, 5]. MySQL, MariaDB и PostgreSQL используются для высоконагруженных систем и синхронизации между серверами [17, 18].

SQL-запросы выполняются через библиотеку JDBI [6]. Она поддерживает работу через аннотации @SqlQuery и @SqlUpdate. Имеет преобразование ответов на запросы в Java-объекты. JDBI обрабатывает транзакции и исключения, автоматически закрывая подключения.

Пул соединений базы данных управляется с помощью HikariCP [15]. Она переиспользует текущее подключение. Следит за жизненным циклом соединений и предотвращает утечки ресурсов, что важно при большом количестве одновременных запросов. HikariCP имеет возможность настроить минимальное и максимальное число соединений, время ожидания и таймауты простоя.

Основной серверной платформой для выполнения плагина выбран Spigot, который предлагает улучшенную реализацию Minecraft сервера [7]. Spigot имеет дополнительный API для обработки игровых событий, включая событие чата AsyncPlayerChatEvent, чтобы асинхронно следить за сообщениями чата. Дополнительным вариантом является платформа Paper, которая еще добавляет оптимизации и расширенный API, совместимый с Spigot API. Обе платформы являются популярными решениями для серверов.

Чтобы обрабатывать сетевые пакеты клиента и сервера Minecraft, используется библиотека PacketEvents [19]. Она имеет готовые обертки пакетов, которые отправляет клиент серверу и наоборот. PacketEvents работает на разных версиях и серверных платформах, включая Spigot, Paper и Fabric. С помощью этой библиотеки можно отправлять собственные пакеты игроку, о которых не будет знать сервер. Все операции выполняются в асинхронных потоках Netty, не используя основной поток сервера.

Обработка асинхронных операций сочетает возможности Spigot API и стандартные средства Java. Планировщик задач Spigot BukkitScheduler выполняет отложенные и периодические операции. Для сложных сценариев асинхронной обработки задействуются CompletableFuture и потоки из пакета java.util.concurrent, распределяя нагрузку по нескольким ядрам процессора [16].

Библиотека Jackson отвечает за управление конфигурацией системы, поддерживая различные форматы. Модуль jackson-dataformat-yaml обеспечивает работу с YAML-форматом, традиционно используемым в конфигурационных файлах Minecraft плагинов. Jackson выполняет автоматическую сериализацию и десериализацию Java-объектов в

конфигурационные файлы, а также поддерживает миграцию настроек при обновлении проекта [11].

Выбранная комбинация Java, Gradle, Google Guice, JDBI, HikariCP и Jackson обеспечивает модульность менеджера сообщений, где каждый компонент отвечает за обработку сетевых пакетов, модерацию контента, форматирование сообщений и управление каналами. Обеспечивается хранение истории модерации, настроек чата и данных игроков с поддержкой пяти СУБД. Каждый компонент подобран с учетом специфических требований, включая необходимость обработки сетевых пакетов и сообщений чата для реализации системы раздельных каналов общения как в Chatty, интеграции с существующими серверными платформами Spigot для обеспечения совместимости с другими плагинами аналогично EssentialsX Chat, и обеспечения кросс-серверной синхронизации для организации единого чата в сетях связанных серверов по аналогии с VentureChat и DiscordSRV.

2.3 Разработка структурной модели разрабатываемой системы

Проектирование системы опирается на модульную архитектуру, в рамках которой ядро управляет всеми компонентами, их инициализацией и взаимодействием. Основное ядро системы, которое управляет всеми остальными компонентами, представлено на рисунке 1.

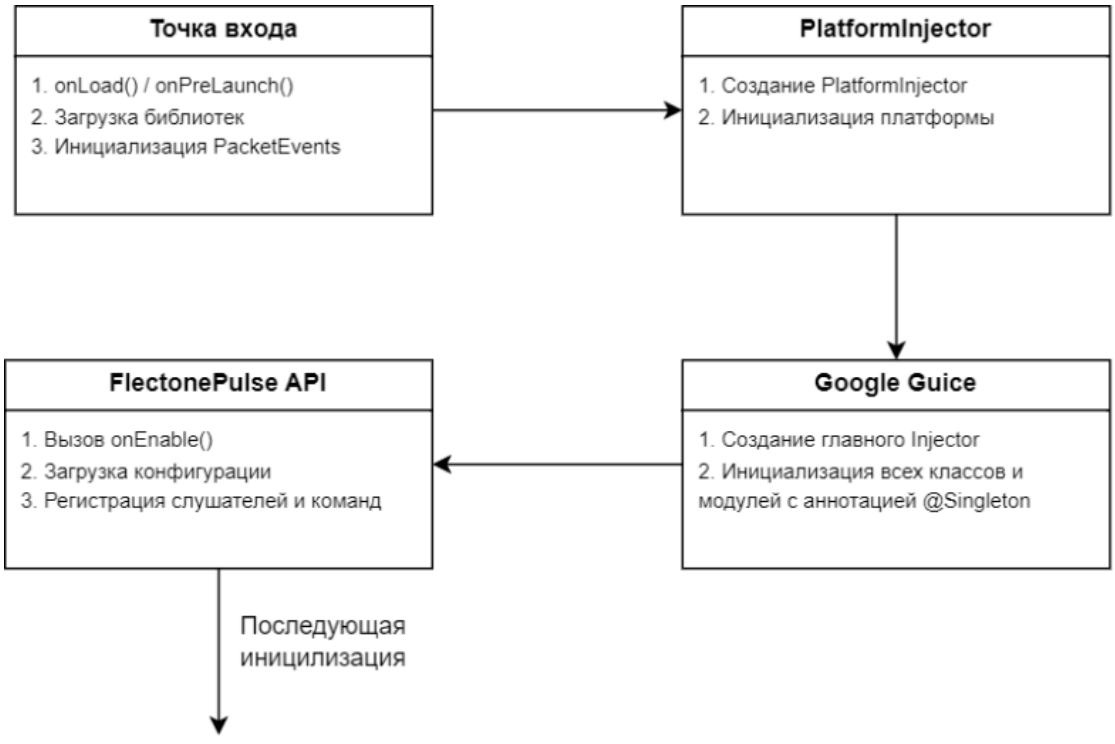

Рисунок 1 – Модель ядра инициализации менеджера сообщений

Как видно из рисунка 1, ядро отвечает за инициализацию системы в точке входа, за создание всех классов с аннотацией @Singleton, загрузку конфигурации, регистрацию слушателей и команд. Оно также обеспечивает взаимодействие между всеми модулями и предоставляет единый API для их интеграции. Кроме того, ядро системы управляет зависимостями между компонентами через фреймворк Google Guice [8].

Для наглядного представления архитектуры системы разработана структурная модель, которая показывает взаимодействие между некоторыми модулями. На рисунке 2 представлена схема, где отображены основные компоненты системы и направления обмена данными между ними.

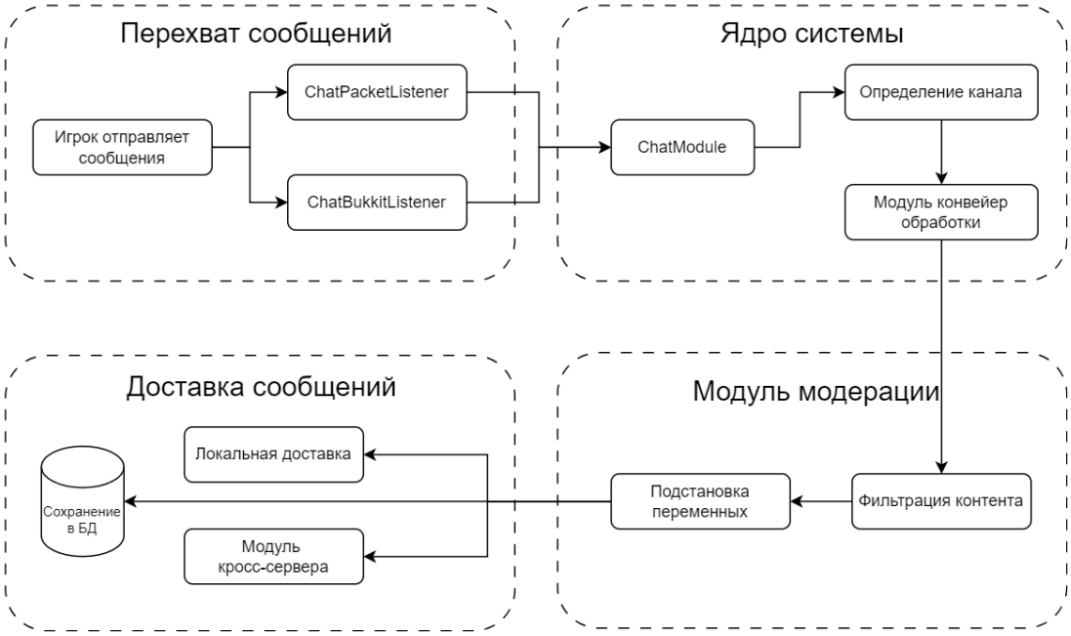

Рисунок 2 – Модель обработки отправленных сообщений

На структурной модели видно, что сообщения игрока проходят несколько этапов обработки. Сначала сырое сообщение перехватывается и передается в ядро системы. Затем ядро определяет соответствующий канал через ChatModule и направляет сообщение через конвейер обработки [20]. После успешного прохождения всех проверок в модуле модерации и форматирования готовое сообщение отправляется обратно игрокам. Все значимые события и данные сохраняются модулем работы с данными в выбранной администратором СУБД. Для сообщений, предназначенных для кросс-серверной рассылки, дополнительно задействуется модуль кросс-сервера, который обеспечивает доставку сообщений на все серверы сети.

Таким образом, структурная модель спроектированной системы демонстрирует четкое разделение ответственности между модулями. Модульная архитектура позволяет гибко настраивать систему под конкретные требования сервера и легко расширять функциональность в будущем.

3. Реализация системы

3.1 Описание алгоритмов и архитектурных особенностей разрабатываемой системы

Разрабатываемая система использует модульную архитектуру, в которой каждый функциональный блок создан в виде независимого компонента, а их связывание осуществляется через фреймворк внедрения зависимостей Google Guice. Такой подход позволяет заменять любой модуль без изменения остальной системы и упрощает тестирование отдельных частей. Центральным элементом выступает ядро, которое при запуске системы последовательно инициализирует все сервисы, загружает конфигурацию и регистрирует слушателей, делегируя всю логику специализированным модулям.

Алгоритм инициализации системы выполняется ядром через вызов метода onEnable. На первом шаге загружаются конфигурационные файлы и настраивается система логирования. Затем устанавливается соединение с базой данных и проверяется ее работоспособность, после чего регистрируются слушатели сетевых пакетов через библиотеку PacketEvents. Далее ядро последовательно активирует модули в порядке их приоритетов. После инициализации всех модулей запускается прокси-регистрация для кросс-серверного взаимодействия и активируется сбор метрик системы. Завершающим шагом ядро рассылает событие завершения запуска, сигнализируя всем компонентам о полной готовности системы к работе. Такой порядок гарантирует, что каждый модуль начинает функционирование только при наличии всех необходимых ему зависимостей, а возникновение ошибки на любом этапе приводит к остановке системы с записью ошибки.

Архитектура системы включает несколько основных модулей, охватывающих перехват сетевых пакетов, управление каналами, конвейерную обработку сообщений, модерацию и работу с данными. Взаимодействие между ними организовано так, что перехват чат-пакетов выполняется асинхронно в потоках Netty, предотвращая блокировку основного игрового цикла, тогда как координация внутри ядра и цепочка обработчиков выполняются синхронно.

Алгоритм обработки входящего сообщения основан на последовательном применении цепочки фильтров, реализованных в виде независимых обработчиков (рисунок 3). При получении сообщения ядро создает контекст, в который входит текст, идентификатор отправителя и канал получателя. Контекст передается первому обработчику в цепочке. Каждый обработчик анализирует содержимое, чтобы применить собственное преобразование или пропустить его. Если нарушений в сообщении нет, контекст передается следующему обработчику. При обнаружении нарушения обработчик останавливается и отправляет сообщение отправителю с ошибкой. Когда все обработчики пройдены, итоговый контекст передается в модуль форматирования. Здесь сообщение собирается в итоговый формат и

отправляется получателям, используя правила видимости канала. Администратор может отключать или включать обработчики через конфигурационный файл без изменения исходного кода. Операции, не требующие сразу ответа, такие как запись истории модерации или обновление статистики игрока, выносятся в отложенное выполнение, используя планировщик задач.

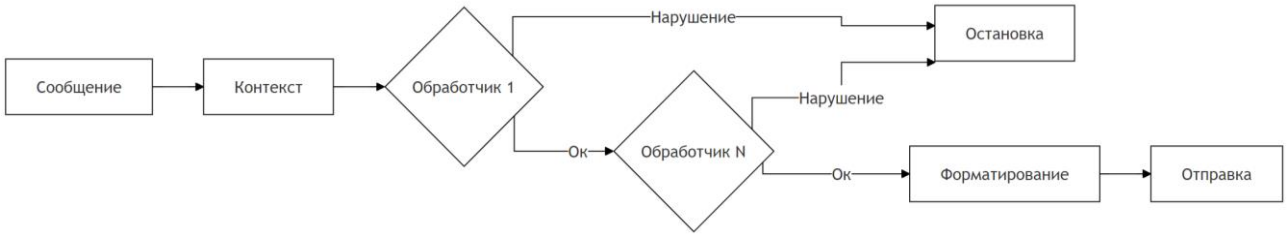

Рисунок 3 – Алгоритм обработки входящего сообщения

В системе есть механизм кросс-серверной синхронизации, реализованный через обмен сообщениями прокси-сервера на базе Velocity или BungeeCord. В отличие от прямых HTTP-вызовов, характерных для микросервисных веб-приложений, здесь синхронизация состояния чата между серверами сети опирается на асинхронную передачу уже сформатированных сообщений, не дублируя логику обработки на каждом узле. Это позволяет игрокам, распределенным по разным серверам, общаться в едином пространстве каналов с минимальными задержками, а согласованность данных поддерживается локальным кэшем. Однако такой подход имеет серверное ограничение. Сообщение может быть доставлено на конкретный сервер только при условии, что на нем присутствует хотя бы один игрок, способный принять передаваемые пакеты, в противном случае синхронизация серверов невозможна.

Архитектура хранения данных построена на абстрактном слое доступа, поддерживающем пять реляционных СУБД. Для разработки и тестирования применяется встроенная H2, не требующая отдельного процесса, тогда как на крупных публичных серверах администратор может переключиться на MySQL или PostgreSQL изменением одного параметра в конфигурации. Абстракция реализована через библиотеку JDBI, унифицирующую выполнение запросов и маппинг результатов, а пул соединений HikariCP обеспечивает эффективное управление подключениями.

Таким образом, ключевой архитектурной особенностью системы является конвейерная обработка сообщений, реализованная через цепочку фильтров. Это решение дополняется абстрактным слоем доступа к данным и кросс-серверной синхронизацией через прокси-сообщения, что в совокупности обеспечивает гибкость и адаптируемость системы.

3.2 Реализация системы и описание этапов разработки

Первым этапом разработки является настройка окружения системы, которая представляет собой модульную структуру, собранную с помощью Gradle. Для достижения совместимости с различными версиями серверов используется плагин jvmdowngrader, который компилирует исходный код под Java 25 и затем понижает его до Java 17 для обеспечения совместимости с ранними версиями серверов Minecraft.

```
plugins {
   id 'xyz.wagyourtail.jvmdowngrader' version '1.3.6'
}

downgradeJar {
    quiet = true
    downgradeTo = JavaVersion.VERSION_17
}
```

Корневой файл build.gradle описывает подпроекты core, bukkit, fabric, velocity, bungeecord и другие. Каждый подпроект отвечает за отдельную платформенную реализацию, что позволяет собирать разные артефакты для разных сред выполнения без дублирования кода. Зависимости объявлены в блоке subprojects, где перечислены общие библиотеки Google Guice, JDBI, HikariCP и Jackson.

```
subprojects {
   repositories {
      mavenCentral()
   }
   dependencies {
      api "org.jspecify:jspecify:$jspecify_version"
      compileOnlyApi "org.mapstruct:mapstruct:$mapstruct_version"
      compileOnlyApi "org.projectlombok:lombok:$lombok_version"
      compileOnly "com.google.inject:guice:$guice_version"
      compileOnly "net.kyori:adventure-api:$adventure_api"
      compileOnly "net.kyori:adventure-text-serializer-legacy:$adventure_api"
      compileOnly "net.kyori:adventure-text-minimessage:$adventure_api"
      compileOnly "net.kyori:adventure-text-serializer-gson:$adventure_api"
      compileOnly "net.kyori:adventure-text-serializer-plain:$adventure_api"
      compileOnly "it.unimi.dsi:fastutil:$fastutil_version"
   }
}
```

Для загрузки внешних библиотек во время выполнения используется Libby, что позволяет не включать все зависимости в итоговый JAR-файл, а подгружать их по необходимости при запуске (рисунок 4).

Рисунок 4 – Загрузка библиотек во время выполнения

Следующим этапом стала реализация точки входа для серверов на базе Bukkit. В методе onLoad происходит ранняя инициализация. Настраивается логгер для вывода информации в консоль. BukkitLibraryResolver загружает необходимые библиотеки из указанных Maven-репозиториев. PacketEvents инициализируется для перехвата сетевых пакетов. Создается инжектор Google Guice с платформенными привязками через BukkitInjector.

```
@Getter
@Singleton
public class BukkitFlectonePulse extends JavaPlugin implements FlectonePulse {

    private FLogger fLogger;
    private LibraryResolver libraryResolver;
    private Injector injector;

    @Override
    public void onLoad() {
        // initialize custom logger
        fLogger = new FLogger(
                logRecord -> this.getLogger().log(logRecord),
                () -> injector == null ? null : injector.getInstance(FileFacade.class)
        );
        fLogger.logEnabling();

        // set up library resolver for dependency loading
        libraryResolver = new BukkitLibraryResolver(this);
        libraryResolver.addLibraries();
        libraryResolver.resolveRepositories();
        libraryResolver.loadLibraries();
```

```
        // configure packetevents api
        System.setProperty("packetevents.nbt.default-max-size", "2097152");
        PacketEvents.setAPI(SpigotPacketEventsBuilder.build(this));

    PacketEvents.getAPI().getSettings().reEncodeByDefault(false).checkForUpdates(false).deb
    ug(false);

        // create guice injector for dependency injection
        injector = Guice.createInjector(Stage.PRODUCTION, new BukkitInjector(this, this,
    libraryResolver, fLogger));

        PacketEvents.getAPI().load();
    }

    }
```

В методе onEnable управление передается в FlectonePulseAPI, где запускаются все сервисы и зарегистрированные модули.

```
@SneakyThrows
public void onEnable() {
    if (!instance.isReady()) return;

    // get configs
    FileFacade fileFacade = instance.get(FileFacade.class);

    // get fLogger
    FLogger fLogger = instance.get(FLogger.class);

    // log plugin information
    fLogger.logDescription();

    // load platform localizations
    instance.get(TranslationService.class).reload();

    // init command registry
    instance.get(CommandRegistry.class).init();

    // register default listeners
    instance.get(ListenerRegistry.class).registerDefaultListeners();

    // setup filter
    instance.get(LogFilter.class).setFilters(fileFacade.config().logger().filter());

    // test database connection
    instance.get(Database.class).connect();

    // initialize packetevents
    instance.initPacketAdapter();

    // reload modules and their children
    instance.get(ModuleController.class).reload();
```

```
        // reload fplayer service
        instance.get(FPlayerService.class).reload();

        // enable proxy registry
        instance.get(ProxyRegistry.class).onEnable();

        // reload metrics service if enabled
        if (fileFacade.config().metrics().enable()) {
            instance.get(MetricsService.class).reload();
        }

        // call enable event
        instance.get(EventDispatcher.class).dispatch(new EnableEvent(instance));

        // log plugin enabled
        fLogger.logEnabled();
    }
```

Метод onDisable выполняет корректное завершение работы, освобождая ресурсы и сохраняя данные.

```
    public void onDisable() {
        instance.terminateFailedPacketAdapter();

        if (!instance.isReady()) return;

        FLogger fLogger = instance.get(FLogger.class);

        // log plugin disabling
        fLogger.logDisabling();

        // call disable event
        instance.get(EventDispatcher.class).dispatch(new DisableEvent(instance));

        // disable task scheduler
        instance.get(TaskScheduler.class).shutdown();

        // close all open inventories
        instance.closeUIs();

        // get fplayer service
        FPlayerService fPlayerService = instance.get(FPlayerService.class);

        // update and clear all fplayers
        fPlayerService.getOnlineFPlayers().forEach(fPlayerService::clearAndSave);
        fPlayerService.clear();

        // disable all modules
        instance.get(ModuleController.class).terminate();

        // unregister all listeners
        instance.get(ListenerRegistry.class).unregisterAll();
```

```
    // terminate packetevents
    instance.terminatePacketAdapter();

    // disable proxy registry
    instance.get(ProxyRegistry.class).onDisable();

    // disconnect from database
    instance.get(Database.class).disconnect();

    // log plugin disabled
    fLogger.logDisabled();
}
```

Инъекция зависимостей реализована через фреймворк Google Guice. Все основные компоненты проекта объявлены с аннотацией @Singleton, что гарантирует создание единственного экземпляра каждого сервиса. Модули инициализации собраны в классе PlatformInjector, где описываются привязки интерфейсов к их реализациям. Интерфейс MessageSender привязан к BukkitMessageSender для платформы Bukkit. Такой подход позволяет легко заменять реализации компонентов при портировании на другие платформы.

```
@Override
protected void configure() {
    bind(FLogger.class).toInstance(fLogger);
    bind(FlectonePulseAPI.class).asEagerSingleton();
    bind(LibraryResolver.class).toInstance(libraryResolver);

    ReflectionResolver reflectionResolver = new ReflectionResolver(libraryResolver);
    bind(ReflectionResolver.class).toInstance(reflectionResolver);

    bind(MessageSender.class).to(BukkitMessageSender.class);

    …
}
```

Третьим этапом реализована модульная система, которая построена на интерфейсе ModuleSimple для всех модулей. Каждый модуль содержит методы onEnable и onDisable. Управление дочерними модулями осуществляется через метод childrenBuilder.

```
public interface ModuleSimple {

    ModuleName name();

    EnableSetting config();

    PermissionSetting permission();

    default void onEnable() {
    }
```

```
    default void onDisable() {
    }

    default BiPredicate<FEntity, Boolean> disablePredicate() {
        return (fEntity, aBoolean) -> false;
    }

    default ImmutableSet.Builder<@NonNull Class<? extends ModuleSimple>>
childrenBuilder() {
        return ImmutableSet.builder();
    }

    default ImmutableSet.Builder<@NonNull PermissionSetting> permissionBuilder() {
        return ImmutableSet.<PermissionSetting>builder().add(permission());
    }

}
```

Модуль сообщений обрабатывает отправку текста в чат. Модуль модерации проверяет сообщения на наличие запрещенных слов. Модуль интеграции с Discord пересылает сообщения между игровым сервером и текстовыми каналами. Включение и отключение модулей происходит через конфигурационный файл, а загрузка модулей выполняется автоматически при старте плагина.

```
    public void handleChatEvent(FPlayer fPlayer, String eventMessage, Runnable cancelEvent,
BiConsumer<String, Boolean> successEvent) {
        if (muteSender.sendIfMuted(fPlayer)) {
            cancelEvent.run();
            return;
        }

        if (disableSender.sendIfDisabled(fPlayer, fPlayer, name())) {
            cancelEvent.run();
            return;
        }

        Chat playerChat = getPlayerChat(fPlayer, eventMessage);
        if (playerChat.config() == null || !playerChat.config().enable()) {
            messageDispatcher.dispatchError(this,
EventMetadata.<Localization.Message.Chat>builder()
                    .sender(fPlayer)
                    .format(Localization.Message.Chat::nullChat)
                    .build()
            );

            cancelEvent.run();
            return;
        }
```

```
            if (cooldownSender.sendIfCooldown(fPlayer, playerChat.cooldown(),
this.getClass().getName() + playerChat.name())) {
                cancelEvent.run();
                return;
            }

            String trigger = playerChat.config().trigger();
            if (!StringUtils.isEmpty(trigger) && eventMessage.startsWith(trigger)) {
                eventMessage = eventMessage.substring(trigger.length()).trim();
            }

            Range chatRange = playerChat.config().range();

            successEvent.accept(eventMessage, playerChat.config().cancel());

            sendMessage(fPlayer, eventMessage, playerChat);
        }
```

Следующим этапом реализована обработка сообщений на асинхронной цепочке событий. Каждое входящее сообщение проходит через последовательность фильтров. Обработчик модерации проверяет текст на наличие запрещенных слов с использованием регулярных выражений и словарей. Обработчик замены подставляет плейсхолдеры, такие как <player>, <world> или <afk>. Обработчик форматирования преобразует цветовые коды из устаревшего формата & в современные теги MiniMessage.

```
        @Singleton
        @RequiredArgsConstructor(onConstructor = @__(@Inject))
        public class EventDispatcher {

            private final ListenerRegistry listenerRegistry;

            @CheckReturnValue
            @SuppressWarnings("unchecked")
            public <T extends Event> T dispatch(Map<Event.Priority,
List<UnaryOperator<Event>>> listeners, T event) {
                if (listeners == null) return event;

                T currentEvent = event;

                for (Event.Priority priority : Event.Priority.values()) {
                    List<UnaryOperator<Event>> handlersList = listeners.get(priority);
                    if (handlersList != null) {
                        for (UnaryOperator<Event> handler : handlersList) {
                            currentEvent = (T) handler.apply(currentEvent);
                        }
                    }
                }

                return currentEvent;
            }
```

```
        @CheckReturnValue
        public <T extends Event> T dispatch(T event) {
            return dispatch(listenerRegistry.getPulseListeners().get(event.getClass()), event);
        }

    }

    public Component build(MessageContext context) {
        // no need to build empty message
        if (StringUtils.isEmpty(context.message())) return Component.empty();

        MessageFormattingEvent event = eventDispatcher.dispatch(new
MessageFormattingEvent(context));
        MessageContext eventContext = event.context();

        try {
            return miniMessage.deserialize(
                    // always need to replace legacy § with & to avoid MiniMessage problems
                    Strings.CS.replace(eventContext.message(), "§", "&"),
                    eventContext.tagResolver()
            );
        } catch (Exception e) {
            fLogger.warning(e);
        }

        return Component.empty();
    }
```

Конфигурация проекта хранится в YAML-файлах с помощью Jackson. Файл config.yml содержит основные настройки базы данных, языка и режима прокси. Файл message.yml определяет правила отправки сообщений, включая радиус действия, место отображения, звуковое сопровождение и задержки. Файл command.yml настраивает команды и их псевдонимы. Файл integration.yml управляет подключениями к внешним сервисам. Локализация вынесена в отдельные файлы в папке localizations, где для каждого языка можно задать собственные форматы сообщений. Чтение конфигурации выполняется при старте плагина и при выполнении команды перезагрузки.

```
    public void reload() throws IOException {
        fileLoader.init();

        // this is to check FlectonePulse version
        // maybe in the future we should put version in a separate file, but I think it's not so
important
        preInitVersion = fileLoader.loadAndMergeConfig(files).version();
        boolean versionChanged = !preInitVersion.equals(BuildConfig.PROJECT_VERSION);

        // backup if version changed
        if (versionChanged) {
            backupFiles(preInitVersion);
```

```
    }

    // load local files
    updateFiles();

    // migrate if version changed
    if (versionChanged) {
        migrateFiles(preInitVersion);
    }

    saveFiles();

    // fix migration problems
    if (versionChanged) {
        updateFiles();
    }
}
```

Командная система реализована на фреймворке Cloud с помощью интерфейса ModuleCommand и контроллера ModuleCommandController. В методе registerCommand происходит регистрация команды. Указываются имя, псевдонимы и обработчик выполнения. Проверка прав доступа выполняется через PermissionChecker. Для каждого аргумента команды добавляется подсказка через метод addPrompt, что позволяет отображать локализованные названия параметров.

```
public void registerCommand(ModuleCommand<?> command,
                            UnaryOperator<Command.Builder<FPlayer>> builder) {
    List<String> aliases = command.config().aliases();
    String commandName = getCommandName(command);

    commandRegistryProvider.get().registerCommand(manager ->
            builder.apply(manager.commandBuilder(commandName, aliases,
CommandMeta.empty())).handler(command)
    );
}

public String addPrompt(ModuleCommand<?> command,
                        int index,
                        Function<Localization.Command.Prompt, String> promptLocalization) {
    List<String> prompts = getPrompts(command);

    // this prompt already registered
    if (prompts.size() > index) {
        return prompts.get(index);
    }

    Class<? extends ModuleSimple> commandClass =
moduleController.getRoot(command.getClass());
    String prompt =
promptLocalization.apply(fileFacade.localization().command().prompt());
```

```
    if (prompts.isEmpty()) {
        commandPromptsMap.put(commandClass, List.of(prompt));
    } else {
        List<String> newPrompts = new ArrayList<>(prompts);
        newPrompts.add(prompt);

        commandPromptsMap.put(commandClass, List.copyOf(newPrompts));
    }

    return prompt;
}
```

Пятым этапом организована работа с базами данных через JDBI. Пул соединений HikariCP управляет подключениями к выбранной СУБД. В конфигурации можно указать тип базы данных, хост, порт, имя базы, логин и пароль. При первом запуске плагин автоматически создает необходимые таблицы. Таблицы fp_player, fp_setting и fp_player_fcolor хранят настройки каждого игрока, такие как имя, айпи, выбранный цвет чата и отключенные уведомления. Таблица fp_moderation содержит историю действий модераторов. Все запросы выполняются асинхронно, чтобы не задерживать основной поток.

```
public void connect() throws IOException {
    downloadDriver();

    HikariConfig hikariConfig = createHikariConfig();
    dataSource = new HikariDataSource(hikariConfig);
    jdbi = Jdbi.create(dataSource);
    jdbi.installPlugin(new SqlObjectPlugin());

    setupTemplateEngine();

    jdbi.registerRowMapper(ConstructorMapper.factory(FColor.class));
    jdbi.registerRowMapper(ConstructorMapper.factory(FPlayerDAO.PlayerInfo.class));
    jdbi.registerRowMapper(ConstructorMapper.factory(Ignore.class));
    jdbi.registerRowMapper(ConstructorMapper.factory(Mail.class));
    jdbi.registerRowMapper(ConstructorMapper.factory(Moderation.class));
    jdbi.registerRowMapper(ConstructorMapper.factory(PlayTime.class));

    executeSQLFile(platformServerAdapter.getResource("sqls/" +
config().type().name().toLowerCase() + ".sql"));
    checkMigration();
    init();
}
```

Финальным этапом разработки подключены интеграции с внешними платформами через отдельные модули. Модуль Discord использует библиотеку Discord4J для подключения к серверу Discord. В файле конфигурации указывается токен бота и идентификаторы каналов. При получении сообщения из Discord модуль преобразует его в игровой формат и отправляет в чат. При

получении сообщения из игры модуль отправляет его в соответствующий канал Discord. Аналогично работают модули Telegram и Twitch. Все интеграции загружаются условно только при наличии соответствующих зависимостей в classpath.

```
public class DiscordIntegration implements FIntegration {

    public void sendMessage(FEntity sender, String messageName, UnaryOperator<String>
discordString) {
        if (gateway == null) return;

        List<String> channels = config().messageChannel().get(messageName);
        if (channels == null) return;
        if (channels.isEmpty()) return;

        channels.forEach(string -> {
            Optional<Snowflake> channel = parseSnowflake(string);
            if (channel.isEmpty()) return;

            Localization.Integration.Discord localization =
fileFacade.localization().integration().discord();
            Localization.Integration.Discord.ChannelEmbed channelEmbed =
localization.messageChannel().getOrDefault(messageName, new
Localization.Integration.Discord.ChannelEmbed("<final_message>", null, null, null));
            sendMessage(sender, channel.get(), channelEmbed, discordString);
        });
    }

}
```

Таким образом, в ходе реализации были выполнены все этапы разработки системы. Настроено сборочное окружение на базе Gradle с применением плагина jvmdowngrader для обеспечения совместимости с разными версиями серверов Minecraft. Создана точка входа с системой внедрения зависимостей через Google Guice. Разработана модульная архитектура на интерфейсе ModuleSimple, позволившая гибко включать и отключать компоненты через конфигурационные файлы. Реализован конвейер асинхронной обработки сообщений, а также абстрактный слой доступа к данным с поддержкой пяти реляционных СУБД и управлением соединениями через HikariCP. На финальном этапе осуществлена интеграция с внешними платформами Discord и Telegram.

3.3 Тестирование системы

После завершения разработки всех ключевых модулей менеджера сообщений был проведен комплексный цикл тестирования, охватывающий проверку функциональности, производительности под нагрузкой и совместимость с различными серверными платформами. Тестирование проводилось на нескольких этапах, каждый из которых позволил выявить и устранить различные классы ошибок.

Первым этапом стало функциональное тестирование каждого модуля в отдельности. Проверялась корректная работа всех типов сообщений на рисунке 5, включая сообщения о входе и выходе игроков, достижениях, смерти, а также обычные текстовые сообщения. Каждый тип сообщений тестировался с разными настройками форматирования, чтобы убедиться в правильной обработке цветовых кодов, тегов MiniMessage и пользовательских текстур. При тестировании упоминаний игроков обнаружилась ошибка проверки статуса. Уведомления отправлялись даже тем, кто уже вышел с сервера.

Рисунок 5 – Функциональное тестирование каждого модуля

Права доступа к каналам тестировались для игроков с привилегиями и без них, система правильно направляет сообщения. Модуль модерации правильно проверяет слова из черного списка и сообщает о нарушениях.

Отдельно проводилось стресс-тестирование производительности системы, чтобы проверить работу при нагрузке. В качестве инструмента для создания ботов применялся фреймворк Mineflayer, позволяющий управлять поведением большого количества фейковых игроков.

```
function createBot(index) {
  const botName = `${config.prefix}${index}`;

  const bot = mineflayer.createBot({
    host: config.host,
    port: config.port,
    username: botName,
    version: false,
    viewDistance: 'tiny',
    chatLengthLimit: 256,
    checkTimeoutInterval: 1000 * 30,
    keepAlive: true
  });

  bot.on('spawn', () => {
    console.log(`[${botName}] подключился`);

    const chatInterval = () => {
      if (!bot.entity) return;
      bot.chat(`Привет от ${botName} - ${Date.now()}`);
      setTimeout(chatInterval, Math.random() * 4000 + 1000);
    };
    setTimeout(chatInterval, Math.random() * 5000);

    const commandInterval = () => {
      if (!bot.entity) return;

      let cmd = commands[Math.floor(Math.random() * commands.length)];

      if (cmd.includes('{target}')) {
        const target = `${config.prefix}${Math.floor(Math.random() * config.botCount)}`;
        cmd = cmd.replace('{target}', target);
      }
      bot.chat(cmd);
      console.log(`[${botName}] команда: ${cmd}`);

      setTimeout(commandInterval, Math.random() * 8000 + 2000);
    };
    setTimeout(commandInterval, Math.random() * 8000);
  });


  return bot;
}
```

Тестирование проводилось локально, где каждый бот подключался к серверу, после чего начинал отправлять сообщения в общий чат с задержкой между сообщениями. Для сбора метрик производительности использовался плагин Spark, измерявший время тиков сервера, загрузку процессора и потребление памяти, а также приложение VisualVM.

Результаты стресс-тестирования представлены на рисунках 6 и 7. Система справляется с нагрузкой без критических задержек. Загрузка центрального процессора возрастала линейно с увеличением числа игроков, но оставалась в пределах допустимых значений. Потребление оперативной памяти также увеличивалось пропорционально количеству ботов. Важно отметить, что все операции по обработке сообщений выполняются асинхронно, основной поток сервера не блокируется. На основе результатов тестирования были внесены оптимизации в конфигурацию и были исправлены проблемы запросов к базе данных.

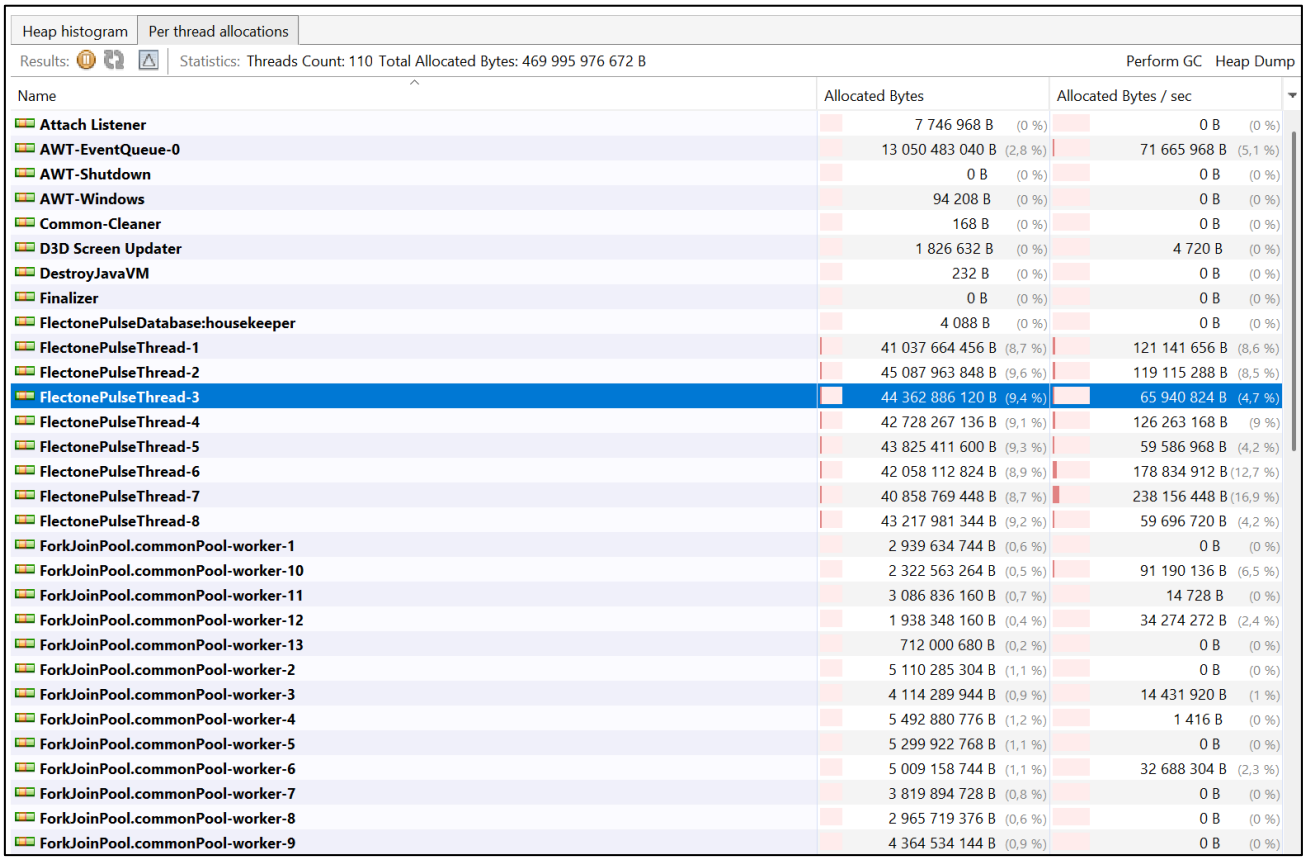

Рисунок 6 – Распределение памяти по каждому потоку

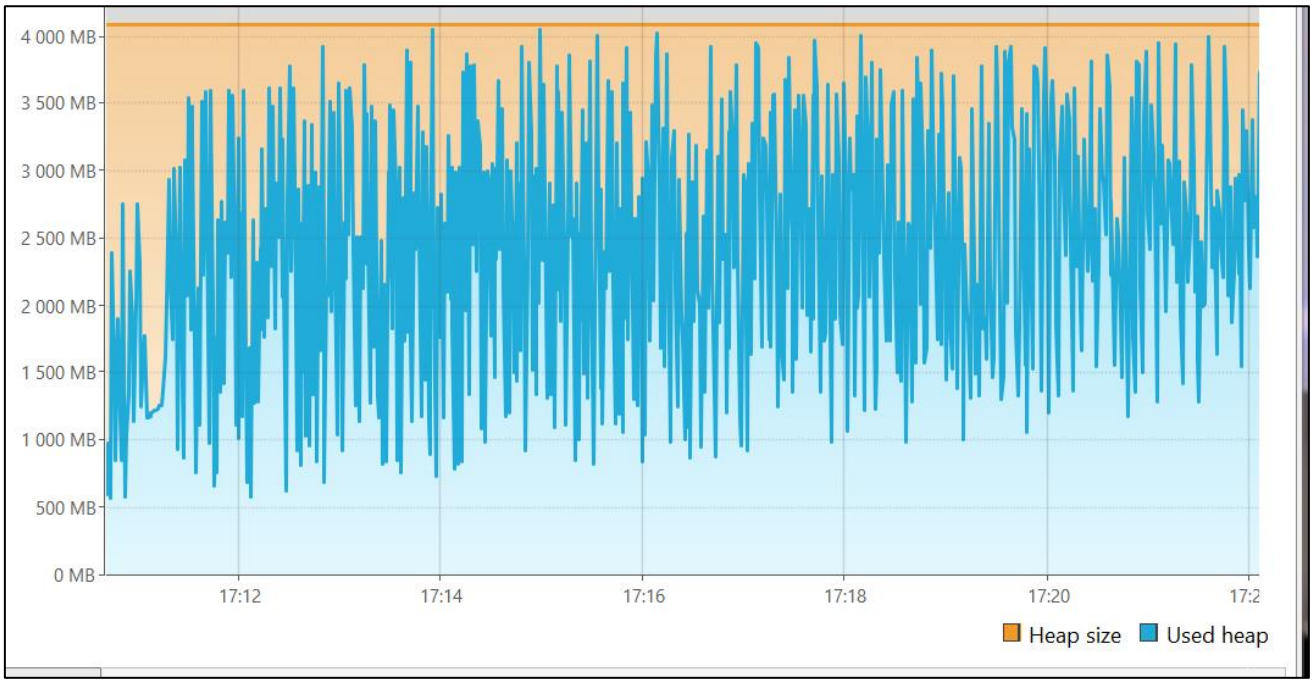

Рисунок 7 – Общая статистика памяти и загрузки сервера

Третьим этапом тестирования стало развертывание проекта на тестовых серверах сообщества и сбор обратной связи от реальных пользователей, которая представлена на рисунке 8. Плагин распространяется через несколько платформ: SpigotMC, CurseForge, Polymart, Modrinth и Hangar. Для систематизации сообщений об ошибках в репозитории проекта добавлены шаблоны для создания баг-репортов. Благодаря обратной связи от администраторов серверов были выявлены проблемы, которые не проявлялись при локальном тестировании. Например, ошибка с неверным ключом в сообщениях о достижениях была обнаружена только после установки плагина на сервере с модифицированной версией Minecraft. Проблема с двойным экранированием амперсандов в URL-адресах проявилась на серверах, использующих нестандартные платформы. Ошибка обработки пиксельных символов в объектном модуле была выявлена при использовании определенных ресурспаков. Все эти отчеты от пользователей были обработаны, и соответствующие исправления включены в последующие версии.

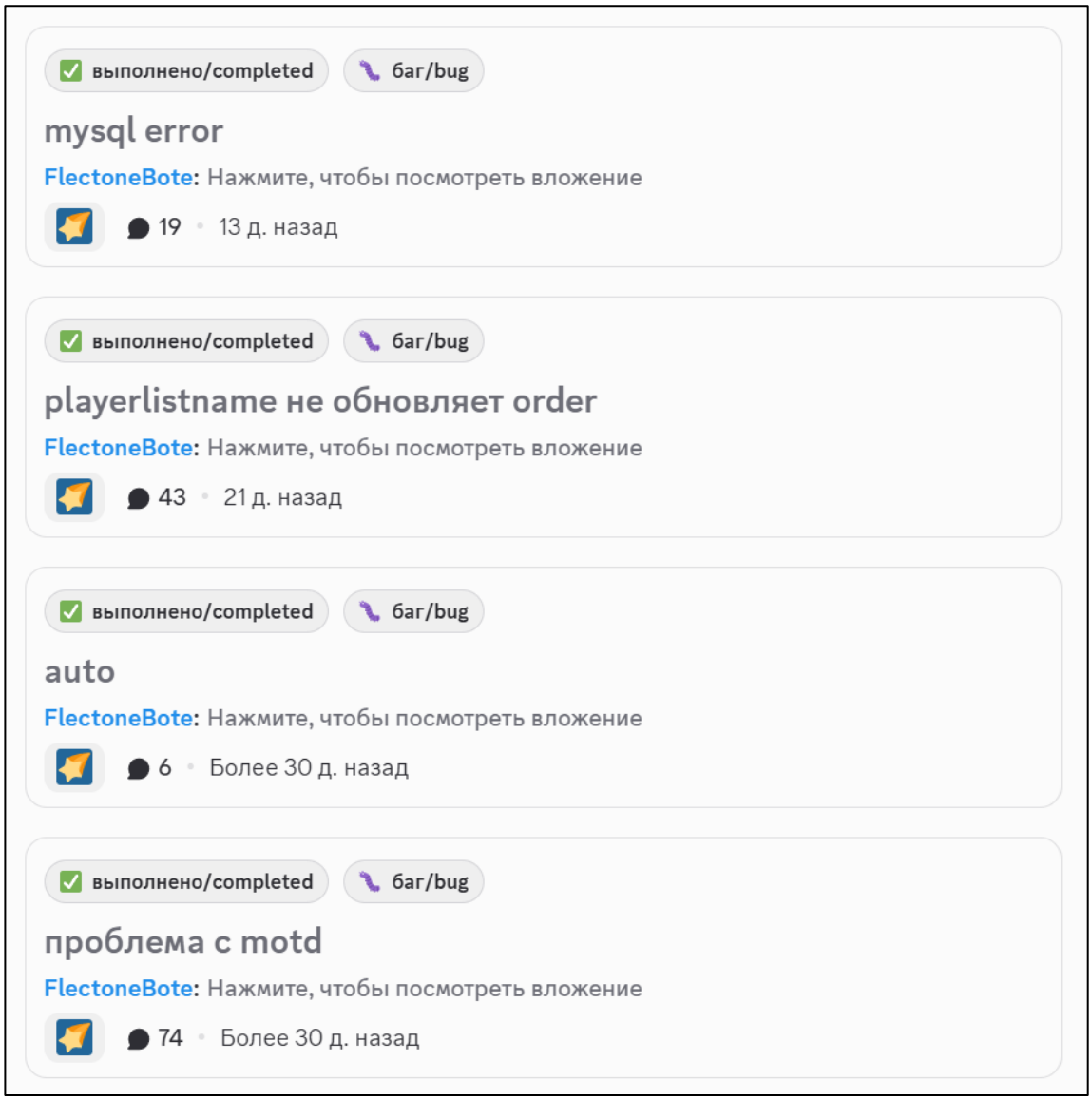

Рисунок 8 – Обратная связь от пользователей проекта

Дополнительно проводилось тестирование совместимости с различными версиями Minecraft. Плагин поддерживает версии от 1.8.8 до 26.1.2, что требует проверки на каждой платформе. В процессе тестирования была выявлена проблема с сообщениями в action bar для версий 1.9 и старше, которая была исправлена добавлением специальной логики совместимости. Также были обнаружены проблемы с перезагрузкой плагина на старых версиях ядра Bukkit и Spigot, что приводило к вылету игроков с сервера.

Заключение

В ходе выполнения выпускной квалификационной работы были изучены теоретические основы разработки менеджера сообщений для игры Minecraft, спроектирована его архитектура, а затем реализована и протестирована система. Проект получил название FlectonePulse, а его исходный код опубликован на GitHub. Были рассмотрены принципы перехвата сетевого трафика между клиентом и сервером, методы фильтрации текстовых сообщений и способы организации кросс-серверного взаимодействия.

На первом этапе были изучены зарубежные и отечественные литературные источники. Для исследования отобраны работы «Large scale in-game spectating system for Minecraft», «Distributed Server for the Game Minecraft», «Разработка сессионного контейнера серверного Java-кода» и «Высокопроизводительный сервер на Netty». В работе «Large scale in-game spectating system» описана система трансляции состояния сервера Minecraft внешним зрителям. «Distributed Server for the Game Minecraft» содержит сведения об использовании функционального программирования для реализации параллельного сервера. «Разработка сессионного контейнера» и «Высокопроизводительный сервер на Netty» предоставили сведения, необходимые для обработки сетевых пакетов и организации высоконагруженных систем в среде Minecraft. Затем проведен анализ существующих аналогов систем управления сообщениями.

Выбраны инструментальные средства, соответствующие критериям масштабируемости, совместимости и модульности. Основным языком программирования выбран Java. Выбор обусловлен прямой совместимостью с официальной серверной реализацией Minecraft и платформой Bukkit. Для низкоуровневой обработки сетевых пакетов выбрана библиотека PacketEvents. Библиотека построена на основе асинхронного Netty и предоставляет API для чтения и модификации пакетов. Серверной платформой выбран Bukkit/Spigot как наиболее распространенная реализация сервера с поддержкой плагинов. Системой сборки выбран Gradle. Для управления зависимостями выбран Google Guice, обеспечивающий внедрение зависимостей в модули. Для работы с базами данных выбраны JDBI и HikariCP.

Разработана структура менеджера сообщений, разделенная на модули перехвата пакетов, ядро системы, управления каналами, цепочки обработчиков, модерации и работы с данными. Ядро системы управляет жизненным циклом остальных модулей и маршрутизацией событий. Цепочка обработчиков последовательно применяет фильтры. Определены взаимодействия между модулями, где каждый компонент выполняет строго определенные функции.

Описаны алгоритмы обработки входящих сообщений и инициализации системы, а также выделены ключевые архитектурные особенности. Алгоритм обработки сообщений построен на последовательном применении цепочки фильтров. На вход подается исходное сообщение и контекст игрока. Каждый фильтр реализует свой метод обработки, возвращающий разрешение или

отклонение. Алгоритм инициализации обеспечивает очередность запуска системы. Сначала загрузка конфигурации и подключение к базе данных, дальше инициализация модулей и регистрация команд. Среди архитектурных особенностей выделены конвейерная модель фильтрации, механизм кросс-серверной синхронизации, а также абстрактный слой доступа к данным с поддержкой пяти реляционных СУБД (MySQL, PostgreSQL, H2, SQLite, MariaDB).

В ходе реализации программного модуля создана система, собранная с помощью Gradle с понижением версии Java для совместимости. Реализованы все ключевые модули, включая обработку сообщений, систему каналов, модерацию, интеграцию с Discord и Telegram. Проведено функциональное тестирование каждого модуля. Функциональное тестирование включало проверку корректности фильтрации, работы команд, сохранения данных. Стресс-тестирование проведено с использованием симуляции игроков, отправляющих сообщения с разным интервалом. Дополнительно тестирование проводилось силами сообщества.

По итогам выпускной квалификационной работы создан менеджер сообщений с открытым исходным кодом. В процессе работы получены навыки проектирования модульной архитектуры с разделением ответственности, реализации перехвата и модификации сетевых пакетов Minecraft. Также получен опыт настройки пула соединений HikariCP и оптимизации запросов через JDBI.

Апробация результатов работы проведена в несколько этапов. Основные положения были представлены на следующих научных конференциях:

– Всероссийской научно-технической конференции «Современные научно-исследовательские и технологические аспекты программной инженерии». Секция: «Современные научно-исследовательские и технологические аспекты программной инженерии» (Оренбург, 18-19 декабря 2025 г.);

– XLVIII Всероссийской студенческой научной конференции. Секция: «Математика и цифровые технологии» (Оренбург, 1-8 апреля 2026 г.).

По теме исследования опубликована научная статья «Проектирование менеджера сообщений для игры Minecraft» [20].

Материалы, подтверждающие апробацию, представлены в приложении А.

Список использованных источников

1 Бальцер, А. Разработка высоконагруженного игрового WebSocket сервера на Java, Netty с поддержкой BattleRoyale/Matchmaking [Электронный ресурс] А. Бальцер // Хабр – русскоязычный веб-сайт в формате системы тематических коллективных блогов с элементами новостного ресурса, 2006-2026 / разраб. Д. Крючков, 2006-2026. – 2023. – URL: https://habr.com/ru/articles/774322/ (дата обращения: 25.04.2026). – Режим доступа: свободный.

2 Виноградов, В. С. Разработка сессионного контейнера серверного Java-кода / В. С. Виноградов // Математические структуры и моделирование. – 2024. – № 2 (30). – С. 77-86.

3 Высокопроизводительный NIO-сервер на Netty [Электронный ресурс] // Хабр – русскоязычный веб-сайт в формате системы тематических коллективных блогов с элементами новостного ресурса, 2006-2026 / разраб. Д. Крючков, 2006-2026. – 2012. – URL: https://habr.com/ru/articles/136456/ (дата обращения: 25.04.2026). – Режим доступа: свободный.

4 Официальная документация H2 Database Engine [Электронный ресурс] // H2 Database – открытая кроссплатформенная реляционная СУБД, 2026. – URL: https://h2database.com/html/main.html (дата обращения: 25.04.2026). – Режим доступа: свободный.

5 Официальная документация SQLite [Электронный ресурс] // sqlite.org – официальный сайт проекта SQLite / разраб. D. R. Hipp, 2026. – URL: https://www.sqlite.org/docs.html (дата обращения: 25.04.2026). – Режим доступа: свободный.

6 Руководство по использованию JDBI [Электронный ресурс] // JDBI Project – библиотека для упрощения взаимодействия с базами данных на языке Java, 2026. – URL: https://jdbi.org/ (дата обращения: 25.04.2026). – Режим доступа: свободный.

7 Спигот API Документация [Электронный ресурс] // SpigotMC – форк серверного мода Bukkit для игры Minecraft, 2026. – URL: https://hub.spigotmc.org/javadocs/spigot/ (дата обращения: 25.04.2026). – Режим доступа: свободный.

8 Фреймворк Google Guice: внедрение зависимостей [Электронный ресурс] // GitHub – онлайн-платформа для хостинга IT-проектов и совместной разработки / разраб. Т. Престон-Вернер, К. Ванстрат, П. Д. Хайетт, С. Чакон, 2007-2026. – URL: https://github.com/google/guice/wiki (дата обращения: 25.04.2026). – Режим доступа: свободный.

9 Bharambe, A. Colyseus: a distributed architecture for online multiplayer games [Электронный ресурс] / A. Bharambe. // Proceedings of the 3rd conference on Networked Systems Design & Implementation (NSDI). – USENIX Association, 2006. – Pp. 155-168. – URL: https://www.usenix.org/event/nsdi06/tech/full_papers/bharambe/bharambe.pdf (дата обращения: 25.04.2026). – Режим доступа: свободный.

10 Delgorge, M. ObserverCraft: a large-scale in-game spectating system for Minecraft-like games [Электронный ресурс] / M. Delgorge. – 2021. – 55 p. – URL: https://jdonkervliet.com/assets/pdf/students/202106-bsc-thesis-milos-delgorge.pdf (дата обращения: 25.04.2026). – Режим доступа: свободный.

11 FasterXML / jackson-docs [Электронный ресурс] // GitHub – онлайн-платформа для хостинга IT-проектов и совместной разработки / разраб. Т. Престон-Вернер, К. Ванстрат, П. Д. Хайетт, С. Чакон, 2007-2026. – URL: https://github.com/FasterXML/jackson-docs (дата обращения: 25.04.2026). – Режим доступа: свободный.

12 Gradle user manual [Электронный ресурс] // docs.gradle.org – официальная справочная документация для инструмента автоматизации сборки Gradle – open-source-решения в разработке ПО / разраб. Gradle Inc, 2010-2026. – URL: https://docs.gradle.org/current/userguide/what_is_gradle.html (дата обращения: 25.04.2026). – Режим доступа: свободный.

13 Herman, T. Distributed server for the game Minecraft [Электронный ресурс] / T. Hejman. – 2023. – 58 p. – URL: https://cyber.felk.cvut.cz/theses/papers/240.pdf (дата обращения: 25.04.2026). – Режим доступа: свободный.

14 Herman, A. E. Transforming Minecraft into a research platform [Электронный ресурс] / A. E. Herman // 2014 IEEE 11th consumer communications and networking conference (CCNC). – ResearchGate – международная онлайн-платформа / Dr. I. Madisch, S. Hofmayer, H. Fickenscher, 2008-2026. – 2014. – URL: https://www.researchgate.net/publication/271472055_Transforming_Minecraft_into_a_research_platform (дата обращения: 25.04.2026). – Режим доступа: свободный.

15 HikariCP documentation [Электронный ресурс] // GitHub – онлайн-платформа для хостинга IT-проектов и совместной разработки / разраб. Т. Престон-Вернер, К. Ванстрат, П. Д. Хайетт, С. Чакон, 2007-2026. – URL: https://github.com/brettwooldridge/HikariCP (дата обращения: 25.04.2026). – Режим доступа: свободный.

16 Java Concurrency Utilities [Электронный ресурс] // docs.oracle.com – официальный справочный центр документации Oracle / разраб. Oracle Corporation, 1993-2026. – URL: https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/ (дата обращения: 25.04.2026). – Режим доступа: свободный.

17 MariaDB documentation [Электронный ресурс] // mariadb.com – основной ресурс для пользователей СУБД MariaDB / разраб. Maria DB Corporation, 2026. – URL: https://mariadb.com/kb/en/documentation/ (дата обращения: 25.04.2026). – Режим доступа: свободный.

18 MySQL 8.4 Reference manual [Электронный ресурс] // dev.mysql.com – официальный сайт документации MySQL / разраб. Oracle Corporation, 1993-2026. – URL: https://dev.mysql.com/doc/refman/8.4/en/introduction.html (дата обращения: 25.04.2026). – Режим доступа: свободный.

19 PacketEvents wiki [Электронный ресурс] // GitHub – онлайн-платформа для хостинга IT-проектов и совместной разработки / разраб. Т. Престон-Вернер, К. Ванстрат, П. Д. Хайетт, С. Чакон, 2007-2026. – URL: https://github.com/

retrooper/packetevents/wiki (дата обращения: 25.04.2026). – Режим доступа: свободный.

20 Симченко, Н. Н. Разработка структурной модели менеджера сообщений для игры Minecraft [Электронный ресурс] / Н. Н. Симченко, А. С. Мочалин // Современные научно-исследовательские и технологические аспекты программной инженерии: материалы Всероссийской научной-методической конференции, Оренбург, 18-19 декабря 2025 г. / Оренбургский государственный университет; ред. А. В. Зайцев. – Оренбург: ОГУ, 2026. – С. 308-312. – URL: https://conference.osu.ru/assets/files/conf_info/conf22/s8.pdf (дата обращения: 23.04.2026). – Режим доступа: свободный.

Приложение А
(обязательное)

Апробация результатов исследования

№ 1362

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное
образовательное учреждение
высшего образования
«Оренбургский государственный университет»
Институт математики и информационных технологий

Сертификат

подтверждает, что

Мочалин
Артём Сергеевич

принял(а) участие
во Всероссийской
научно-технической конференции

«СОВРЕМЕННЫЕ НАУЧНО-ИССЛЕДОВАТЕЛЬСКИЕ
И ТЕХНОЛОГИЧЕСКИЕ АСПЕКТЫ
ПРОГРАММНОЙ ИНЖЕНЕРИИ»

18 – 19 декабря

Проректор
по научной работе С.Н. Летута

Оренбург
2025

Рисунок А.1 – Сертификат участника Всероссийской научно-технической конференции «Современные научно-исследовательские и технологические аспекты программной инженерии»

Рисунок А.2 – Сертификат участника XLVIII Всероссийской студенческой научной конференции


№ 746

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное
образовательное учреждение
высшего образования
«Оренбургский государственный университет
имени В.А. Бондаренко»

Благодарность

НАГРАЖДАЕТСЯ

Мочалин
Артём Сергеевич

студент(ка) группы 22ФИИТ(б)РАИС

за активное участие в XLVIII Всероссийской студенческой научной конференции и достигнутые успехи

Проректор
по научной работе С.Н. Летута

Приказ от 16.04.2026 № 304

Рисунок А.3 – Благодарность участнику XLVIII Всероссийской студенческой научной конференции