МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра параллельных вычислений

Направление подготовки 09.04.01 Информатика и вычислительная техника
Направленность (профиль): Технология разработки программных систем

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

Буйко Кирилла Алексеевича

Тема работы:

РАЗРАБОТКА ПОДСИСТЕМЫ ИНТЕГРАЦИИ КОМПОНЕНТОВ СИСТЕМЫ LUNA НА ОСНОВЕ СОБЫТИЙНОГО ПОДХОДА

«К защите допущена»
Заведующий кафедрой,
д.т.н, профессор
Малышкин В. Э. /……..
(ФИО) / (подпись)
« 30 » мая 2025г.

Руководитель ВКР
к.т.н,
доц. каф. ПВ ФИТ НГУ
Перепёлкин В. А. /……...
(ФИО) / (подпись)
« » мая 2025г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ
Киреев С. Е. /……..
(ФИ О) / (подпись)
« » мая 2025г.

Новосибирск, 2025

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)
Факультет информационных технологий
Кафедра параллельных вычислений
Направление подготовки: 09.04.01 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА
Направленность (профиль): Технология разработки программных систем

УТВЕРЖДАЮ

Зав. кафедрой Малышкин В. Э.
(фамилия, И., О.)

…….
(подпись)
« 08 » 02 2025г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ МАГИСТРА

Студенту Буйко Кириллу Алексеевичу, группы 23222
(фамилия, имя, отчество, номер группы)

Тема: Разработка подсистемы интеграции компонентов системы LuNA на основе событийного подхода
(полное название темы выпускной квалификационной работы магистра)

утверждена распоряжением проректора по учебной работе от 03.11.2023 №0412
скорректирована распоряжением проректора по учебной работе от 21.10.2024 №0383
скорректирована распоряжением проректора по учебной работе от 07.02.2025 №0043
Срок сдачи студентом готовой работы 20 мая 2025 г.
Исходные данные (или цель работы): Разработка подсистемы интеграции компонентов системы LuNA на основе событийного подхода
Структурные части работы: обзор существующих решений, описание решаемой задачи, разработка подсистемы интеграции, реализация подсистемы интеграции, тестирование

Руководитель ВКР
доц. каф. ПВ ФИТ НГУ,
к.т.н.
Перепёлкин В. А. /……..
(ФИ О) / (подпись)

« 08 » февраля 2025г.

Задание принял к исполнению
Буйко К. А. /……..
(ФИО студента) / (подпись)

« 08 » февраля 2025г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ
Киреев С. Е. /……..
(ФИ О) / (подпись)

« 08 » февраля 2025г.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящей выпускной квалификационной работе применяют следующие термины с соответствующими определениями:

gRPC (google Remote Procedure Calls) – система удалённого вызова процедур, использующая HTTP/2 и Protobuf в качестве транспорта

HTTP (Hyper Text Transfer Protocol) – протокол передачи гипертекста, сетевой протокол прикладного уровня

JSON (JavaScript Object Notion) – текстовый формат обмена данными

Proto файлы – описание структуры данных для формата Protobuf

LuNA – система автоматического конструирования численных параллельных программ, разработанная и развиваемая в ИВМиМГ СО РАН на основе концепции активных знаний

ВВЕДЕНИЕ

Современные цифровые экосистемы формируются за счет объединения разнородных компонентов, разработанных с использованием различных технологических платформ, языков программирования и архитектурных парадигм. Такая разнородность порождает фундаментальную проблему: отсутствие унифицированных механизмов взаимодействия между автономными модулями. Различия в протоколах обмена данными, форматах сообщений и правилах обработки запросов существенно затрудняют создание целостных систем, способных функционировать как единое целое. Проблема усугубляется в динамических средах, где требования к адаптивности, масштабируемости и реактивности на внешние события делают традиционные подходы к интеграции неприменимыми. Например, необходимость перераспределения ресурсов в реальном времени, реакция на действия в системе, поддержка потоковой обработки данных или гибкое изменение бизнес-логики системы требуют иных методов координации компонентов.

Создание подсистемы интеграции на основе событийного подхода позволяет решить две ключевые задачи. Во-первых, такая подсистема объединяет части системы, которые используют разные способы обмена данными, даже если их интерфейсы изначально несовместимы. Во-вторых, она делает взаимодействие между компонентами гибким: вместо жёсткой настройки связей, можно настраивать динамические действия и правила в зависимости от событий и состояния системы. Это позволяет легко добавлять новую логику, менять порядок работы компонентов или подключать новые модули без переписывания существующего кода. Кроме того, система становится проще для масштабирования, так как каждый компонент работает автономно, выполняя свою роль в общем процессе. Этот подход имеет потенциал, но недостаточно разработать подсистему интеграции. Необходима система, для которой она будет настроена. В данной работе реализована подсистема интеграции для системы LuNA, которая хорошо подойдёт для демонстрации и тестирования подхода.

Система LuNA – это система конструирования, генерации и исполнения параллельных программ. Процесс её работы разбит на несколько этапов, каждый из которых реализуется в отдельном сервисе разными разработчиками, обладающими уникальными профессиональными навыками. Это приводит к тому, что технологии реализации сильно отличаются. Без связующего элемента и установленных форматов компоненты не смогут передавать данные друг другу, как и пользователю. Также система имеет изменчивый характер, компоненты могут подключаться, заменяться, может обновляться их интерфейс. В системе имеется множество компонентов одного типа (хранилищ значений переменных, исполнителей фрагментов кода и пр.). Необходима возможность добавлять логику в систему, от принятия решения о конечной точке запроса до исполнения задач независимо от действий компонентов.

Цель работы – разработка подсистемы интеграции компонентов системы LuNA на основе событийного подхода. Для достижения цели необходимо решить следующие задачи:

1. Определить форматы передачи данных между компонентами и внутри подсистемы интеграции;

2. Разработать масштабируемую архитектуру подсистемы интеграции;

3. Реализовать подсистему интеграции, включая поддержку компонентов и логику управления запросами;

4. Развернуть подсистему и провести тестирование.

Разрабатываемая система должна удовлетворять следующим требованиям:

1. Передача запросов от компонента к компоненту;

2. Поддержка компонентов, работающих с несогласованным интерфейсом и разными протоколами передачи данных;

3. Потоковая передача данных между компонентами;

4. Возможность управлять запросами, их конечной точкой и содержанием удалённо;

5. Возможность добавления активных модулей, запускающих сценарии независимо от действий системы.

1 ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ

Существует ряд инструментов, методов и подходов, которые имеют назначение, частично схожее с необходимым. Проанализируем их и определим, какие из них можно использовать при разработке решений поставленных задач.

1.1 HTTP

HTTP – это протокол прикладного уровня, предназначенный для передачи структурированных данных между распределёнными системами. Первоначально разработанный для обмена гипертекстовыми документами в рамках Всемирной паутины, HTTP стал основным средством взаимодействия в современных распределённых системах, включая микросервисную архитектуру [1].

1.1.1 Принцип работы

HTTP функционирует по модели «клиент-сервер», где:

– Клиент (например, веб-браузер или микросервис) инициирует соединение и отправляет HTTP-запрос;

– Сервер (например, веб-сервер или другой микросервис) обрабатывает запрос и возвращает HTTP-ответ.

Ключевые компоненты HTTP-сообщения:

– Метод запроса определяет тип операции, как GET и POST на получение и отправку данных;

– URI указывает адрес целевого ресурса (например, /api/users);

– Заголовки (Headers) содержат метаданные;

– Тело сообщения (Body) передаёт полезную нагрузку, обычно в формате JSON или XML [1, 2, 3].

1.1.2 Область применения

HTTP является универсальным протоколом прикладного уровня, который находит применение в различных сферах, включая веб-разработку, облачные вычисления и микросервисную архитектуру. Его ключевая роль заключается в обеспечении стандартизированного обмена данными между клиентами и серверами.

1.1.3 Применимость к поставленной задаче

HTTP подходит для связи компонентов в микросервисной системе. Но его (в чистом виде) в данной работе невозможно использовать, поскольку это исключительно транспортный протокол, не имеющий механизмов и логики. Необходимо учитывать множество факторов: определить, какому компоненту отправить запрос, в каком виде (интерфейсы не консистентные и могут меняться), где этот компонент находится (адрес может меняться). Каждый компонент должен хранить в себе эту логику, базы компонентов и адаптеры компонентов, обновлять их статически. А если в дальнейшем понадобится расширить работу и, например, связать несколько систем активных знаний, то сложно представить схему их взаимодействия на уровне каждого отдельного компонента в целом. Однако, в сочетании с другими инструментами и архитектурами он, благодаря своей простоте и распространённости, может быть успешно применён в качестве транспортного протокола обмена данными.

1.2 gRPC

gRPC (Google Remote Procedure Call) — это высокопроизводительная открытая RPC-платформа, разработанная Google в 2015 году и переданная под управление Cloud Native Computing Foundation. В отличие от традиционных HTTP API, gRPC использует бинарные протоколы и современные технологии, такие как HTTP/2 и Protocol Buffers (Protobuf), для эффективной передачи данных между распределёнными системами [4].

1.2.1 Архитектура

Ключевые компоненты gRPC:

– Protocol Buffers (Protobuf) – язык-независимый формат сериализации данных, используемый для описания интерфейсов (*.proto-файлы) и генерации клиентского/серверного кода [5];

– HTTP/2 – мультиплексируемый транспортный протокол, обеспечивающий двунаправленную потоковую передачу, сжатие заголовков и низкие задержки [6];

– Строго типизированные сервисы – методы и сообщения определяются в .proto-файлах, что исключает несоответствия в API [4].

1.2.2 Принцип работы

Принцип работы gRPC основан на модели удалённого вызова процедур (RPC), где клиент и сервер обмениваются строго типизированными сообщениями через бинарный протокол. Контракты методов и структур данных определяются в proto файлах с помощью Protocol Buffers (Protobuf), после чего компилятор генерирует клиентские и серверные заглушки (stubs) для выбранного языка программирования. Клиент вызывает методы на сервере так, будто они локальные, а gRPC автоматически сериализует запросы в бинарный формат и передаёт их по HTTP/2, обеспечивая мультиплексирование, потоковую передачу и низкие накладные расходы. Сервер десериализует запрос, выполняет логику и возвращает ответ в том же соединении. Поддержка четырёх режимов взаимодействия (унарный, серверный поток, клиентский поток и двунаправленный поток) делает gRPC гибким решением для высоконагруженных и распределённых систем [4].

1.2.3 Область применения

gRPC может применяться в различных областях:

– Микросервисные архитектуры: взаимодействие сервисов с низкой задержкой;

– Мобильные приложения: минимизация трафика и ускорение передачи данных;

– IoT и реальное время: двунаправленные потоки для сенсоров и устройств;

– Облачные вычисления: интеграция облачных сервисов (Google Cloud, AWS).

1.2.4 Применимость к поставленной задаче

gRPC отлично подходит для связи компонентов в микросервисной системе, поскольку быстрее, чем стандартный HTTP/REST API, гарантирует сходимость интерфейсов благодаря строгой декларации типов, единой для всех,

поддерживает больше десяти языков программирования для генерации готового для применения кода со всеми методами для клиента и сервера, а также встроенным механизмам, гарантирующим доставку и отлов ошибок. Можно сказать, микросервисные системы – основная область применения gRPC. Это надёжный вариант, но, как и HTTP, неподходящий. Помимо тех же причин, что у HTTP, имеется дополнительная – сложно объединить все компоненты единой декларацией типов, особенно на стадии разработки. Она может потерять обратную совместимость при каждой итерации доработок, что приведёт к ошибкам при неосмотрительности. Это особенность gRPC – желательно разработать архитектуру и запросы заранее, чтобы модернизация в будущем проходила с минимальными помехами. Но в последствии, после стабилизации системы, можно постепенно мигрировать на него с других протоколов.

1.3 RabbitMQ

RabbitMQ является брокером сообщений, реализующим протокол AMQP и поддерживающим дополнительные стандарты обмена сообщениями. Он функционирует как промежуточное звено между компонентами системы, обеспечивая прием, маршрутизацию и доставку сообщений согласно событийной модели взаимодействия. В этой модели производители (publishers) отправляют сообщения в брокер, который затем распределяет их среди потребителей (consumers) согласно заданным правилам [7, 8].

Брокер поддерживает различные схемы взаимодействия, включая очереди сообщений, публикацию/подписку и RPC-вызовы. RabbitMQ обеспечивает маршрутизацию через механизм обменников (exchanges), которые определяют правила распределения сообщений между очередями [7].

1.3.1 Архитектура

Архитектура RabbitMQ включает следующие основные компоненты:

– Брокер — центральный сервер, управляющий очередями и маршрутизацией сообщений;

– Очереди — структуры данных, хранящие сообщения до их обработки потребителями;

– Обменники — точки входа сообщений в брокер, определяющие правила маршрутизации;

– Связи — правила, связывающие обменники с очередями на основе ключей маршрутизации;

– Продюсеры — приложения, отправляющие сообщения в обменники;

– Консьюмеры — приложения, получающие сообщения из очередей [7].

1.3.2 Принцип работы

Использование RabbitMQ включает следующие этапы:

1. Создание обменника с указанием типа;
2. Объявление очередей и их привязка к обменникам с помощью ключей маршрутизации;
3. Продюсер публикует сообщение в обменник с указанием ключа маршрутизации;
4. Обменник перенаправляет сообщение в соответствующие очереди;
5. Консьюмер подключается к очереди и запрашивает сообщения;
6. После обработки сообщения консьюмер отправляет подтверждение.

1.3.3 Области применения

RabbitMQ применяется в множестве сценариях:

– В микросервисной архитектур — асинхронное взаимодействие сервисов;

– Фоновая обработка задач — отложенное выполнение ресурсоемких операций (например, генерация отчетов);

– Балансировка нагрузки — распределение задач между несколькими обработчиками;

– Интеграция legacy-систем — обмен данными между разнородными системами;

– IoT-решения — обработка событий от устройств с использованием MQTT;

– Логирование и мониторинг — сбор и анализ логов в реальном времени;

1.3.4 Применимость к поставленной задаче

RabbitMQ как брокер сообщений имеет преимущества для данной системы:

1. Позволяет компонентам без статичных адресов подключаться и отправлять сообщения;

2. Компоненты могут слушать сообщения, предназначенные для них и отвечать в отдельные очереди;

3. Асинхронно обрабатывает поступающие запросы и хранит очередь сообщений при неисправностях в компонентах.

Но RabbitMQ не имеет готовой методологии или встроенного решения для покрытия всех требований. Он требует от компонентов работы с чётким интерфейсом взаимодействия, что потребует их переработки. Логика, связанная с работой системы и обработкой запросов (в том числе принятием решения о конечной точке запроса) не может быть реализована без дополнительных протоколов и структур. В очередях сообщений не получится реализовать передачу потоков данных без буферизации и задержек. Сами запросы передаются медленнее в сравнении со стандартными клиент-серверными подходами. Следовательно, даже как основа для разработки решения он подходит с большими допущениями.

1.4 Kong

Kong — это платформа для управления программными интерфейсами приложений, обеспечивающая маршрутизацию, аутентификацию, мониторинг и контроль доступа к микросервисным архитектурам. Разработанный на основе высокопроизводительного прокси-сервера NGINX и базы данных PostgreSQL, Kong предоставляет масштабируемое решение для оркестрации API в облачных и гибридных средах [9, 10, 11].

Современные распределённые системы, построенные на принципах микросервисной архитектуры, требуют эффективного управления API для обеспечения безопасности, отказоустойчивости и наблюдаемости. Kong решает эти задачи, предлагая модульную и расширяемую платформу,

функционирующую как API-шлюз (API Gateway) и слой взаимодействия между клиентами и серверными приложениями [9].

1.4.1 Архитектура

Kong построен по модульной схеме, где базовая функциональность расширяется за счёт плагинов. Основные компоненты включают:

– Kong Core — ядро системы, отвечающее за обработку HTTP-запросов, балансировку нагрузки и маршрутизацию;

– База данных (PostgreSQL, Cassandra) — хранит конфигурации API, плагины и метрики;

– Admin API — RESTful-интерфейс для управления настройками;

– Плагины — модули, добавляющие функциональность (OAuth2, JWT-аутентификация, rate-limiting, логирование, пользовательскую логику) [4].

1.4.2 Основные функции

Kong предоставляет следующие возможности:

– Маршрутизация и балансировка нагрузки — динамическое перенаправление запросов между сервисами на основе URL, заголовков или методов HTTP;

– Аутентификация и авторизация;

– Ограничение скорости запросов — защита от DDoS-атак и злоупотреблений API;

– Мониторинг и аналитика — интеграции с другими инструментами;

– Трансформация запросов — модификация HTTP-заголовков и тела запросов/ответов [9].

1.4.3 Области применения

Kong применяется в следующих сценариях:

– Микросервисные архитектуры — централизованное управление API множества сервисов.

– Гибридные и мультиоблачные среды — обеспечение единой точки входа для API, развёрнутых в разных облаках.

– Legacy-миграция — постепенная модернизация монолитных систем за счёт API-фасада.

– Безопасность — фильтрация трафика, проверка подлинности клиентов.

1.4.4 Применимость к поставленной задаче

Данный инструмент имеет широкие возможности кастомизации, а именно:

– Позволяет динамически добавлять и исключать компоненты;

– Имеет поддержку плагинов, в которых можно реализовать пользовательскую логику, в том числе принятие решения о конечной точке запроса в зависимости от состояния системы;

– Плагины могут выступать в роли адаптеров для поддержки компонентов с не консистентным интерфейсом, преобразовывая запросы в необходимый вид для различных протоколов;

– С помощью плагинов можно реализовать множество систем активных знаний и соединять их динамически.

Хотя Kong предоставляет возможности для построения системы поверх себя, сам по себе он не покрывает все требования. Более того, он лишь обеспечивает подключение компонентов, использующих разные протоколы передачи данных, без преобразования данных при передаче между узлами системы. Его можно использовать как базу для разработки решения, но он может оказаться излишним и неудобным в применении на практике из-за довольно низкоуровневых плагинов, имеющих свои ограничения как по возможностям, так и доступным для использования данным.

1.5 Выводы

Как видно по представленным инструментам, готового решения не найдено. Децентрализованные варианты, такие как использование чистого HTTP и gRPC не подходят из-за неопределённого числа компонентов и невозможности ввести общую логику системы и обработки запросов. Брокер сообщений, RabbitMQ, структурирует поток запросов, удобно передаёт сообщения между компонентами, но не имеет встроенных возможностей для управления потоком запросов и их распределения. Как основа для разработки он не подходит,

поскольку не поддерживает потоковую передачу данных и требует от всех компонентов определённый формат передачи данных. API Gateway, Kong, также не позволяет покрыть все требования «из коробки». Как основа он подходит лучше, чем RabbitMQ, так как даёт возможность влиять на сам запрос, конечную точку и трансформировать его без задержек. Но возникает вопрос к целесообразности, поскольку на плагинах будет намного больше ответственности, чем на самом Kong, которому останется только передать запрос на длинные цепочки плагинов и адаптеров. Следовательно, для данной работы оптимальным вариантом будет разработка решения, не основанного на существующих инструментах, но использующего некоторые из них, чтобы сделать его наиболее удобным и адаптируемым для использования.

2 ОПИСАНИЕ РЕШАЕМОЙ ЗАДАЧИ

Данная работа встраивается в исследование, посвященное разработке технологии активных знаний для автоматического конструирования решений прикладных задач на основе системы LuNA. Эта разработка направлена на решение фундаментальной проблемы автоматизации применения накопленных человечеством программных модулей в новых вычислительных задачах. Ключевой идеей, обеспечивающей автоматическое применение модулей, является идея о выводе решения задачи, формально поставленной на вычислительной модели. Вычислительная модель — это двудольный ориентированный конечный граф, доли которого образуют два множества вершин, называемых множествами операций и переменных. Входящие в операцию и исходящие из неё дуги определяют входные и выходные переменные операции соответственно. Вычислительная модель описывает некоторую предметную область, где свойства объектов предметной области описаны множеством переменных (значения свойств представляются значениями переменных), а возможность вычислять значения одних свойств из других представлена множеством операций (см. пример на рис. 1) [12].

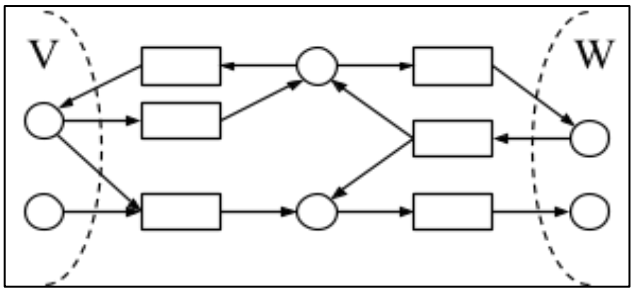

Рисунок 1 – Пример вычислительной модели. Круги — переменные, прямоугольники — операции [12]

Определим на множестве переменных вычислительной модели два подмножества — V и W, и назовём их входными и выходными переменными задачи, зададим значения всех переменных из V. В таком случае будем говорить о том, что поставлена VW-задача на вычислительной модели. Если в вычислительной модели существует подмножество операций, упорядоченное применение которых к заданным значениям переменных позволит получать

значения новых переменных до тех пор, пока все переменные из W не получат значения, то это подмножество будем называть VW-планом. Любая VW-задача может иметь ноль или более VW-планов. Очевидно, что для заданной VW-задачи можно ставить задачу поиска VW-плана, и в случае успеха вычислить значения всех переменных из W, имея на руках значения переменных из V [12].

Такова базовая идея, обеспечивающая возможность автоматического применения существующих модулей для решения новых задач. Каждый модуль, который ставится в соответствие операции, определяет функцию, связывающую входные и выходные переменные операции. Косвенно функционально оказываются связаны и переменные, не связанные операциями непосредственно. Выбор множеств V и W, таким образом, является способом задания функциональной спецификации задачи. Это обеспечивает возможность по функциональной спецификации задачи автоматически строить её решение [12].

Для осуществления автоматического конструирования программ используется множество компонентов. Перечислим некоторые виды компонентов:

- Библиотека фрагментов кода;
- База значений переменных;
- Интерфейс пользователя;
- Транслятор;
- Планировщик;
- Интерпретатор;
- Генератор;
- Профилировщик;
- Исполнитель.

Все компоненты должны обмениваться запросами друг с другом. Например, исполнитель запрашивает значения переменных у базы значений переменных. В работе [12] кратко описано назначение каждого компонента, базовый принцип действия, но как запросы передаются, обрабатываются и доставляются не описано ни в каком виде. Однако, метод их объединения,

очевидно, необходим. И цель данной работы разработать подсистему интеграции для соединения всех компонентов в систему активных знаний. Определим список задач:

1. Определить пространство возможных запросов в системе;
2. Разработать форматы передачи данных запросов подсистемы интеграции;
3. Разработать архитектуру подсистемы интеграции;
4. Реализовать подсистему интеграции;
5. Реализовать адаптеры для подключения сторонних компонентов (база значений переменных и база фрагментов кода);
6. Разработать логику работы для сценариев:
   - Работа с несколькими хранилищами переменных;
   - Учёт качества работы компонентов;
   - Работа с несколькими исполнителями.
7. Реализовать логику системы для сценария работа с несколькими хранилищами переменных;
8. Развернуть подсистему и провести тестирование.

3 РАЗРАБОТКА ПОДСИСТЕМЫ ИНТЕГРАЦИИ

3.1 Исследование запросов в системе

В системе компоненты обмениваются друг с другом запросами. Опишем запросы, которые могут передаваться между компонентами в системе в таблице ниже:

Таблица 1 – Запросы в системе LuNA

Описание	Параметры запроса	Тело запроса	Ответ на запрос
Библиотека фрагментов кода			
Запрос списка фрагментов кода	Нет	Нет	Список фрагментов кода в формате json
Запрос информации о фрагменте кода	Идентификатор фрагмента кода	Нет	Информация о фрагменте кода в формате json
Добавить фрагмент кода	Идентификатор фрагмента кода	Архив в формате tar	Нет
Получить файлы фрагмента кода	Идентификатор фрагмента кода	Нет	Архив в формате tar
Получить обработанный плагином фрагмент кода	Идентификатор фрагмента кода, идентификатор плагина	Нет	Обработанный фрагмент кода
Получить список плагинов ФК	Идентификатор фрагмента кода	Нет	Список поддерживаемых плагинов в формате json
Добавить плагин	Идентификатор плагина	Архив в формате tar	Нет
Хранилище значений переменных			
Получить список значений переменных	Нет	Нет	Массив идентификаторов значений переменных
Получить значение переменной	Идентификатор значения переменной	Нет	Значение переменной

Описание	Параметры запроса	Тело запроса	Ответ на запрос
Добавить значение переменной	Нет	Значение переменной	Идентификатор значения переменной
Удалить значение переменной	Идентификатор значения переменной	Нет	Нет

3.2 Разработка формата передачи данных

Видно, что все запросы делятся на те, которые отправляют данные, требуют исполнения операции и отправляют данные. Первые два можно объединить, поскольку у них схожий формат ответа. Так, все запросы можно разделать на два вида – Get (получение) и Set (отправка/исполнение).

Get запросы включают:

– Тип запроса, наименование, отражающее назначение запроса;

– Информационный объект, состоит из параметров необходимых для получения данных.

В ответ Get запроса возвращается поток данных, содержащий требуемые данные. Set запросы устроены противоположным образом, то есть отправляется поток данных, опционально с ним может отправляться информационный объект для обновления или привязки данных, в ответ возвращается информационный объект с результатом отправки данных, как идентификатор при сохранении значения переменной. По такой схеме можно формализовать каждый из вышеописанных запросов и принимать запросы в подсистеме интеграции от других компонентов системы.

3.3 Разработка управляющих модулей подсистемы

Запросы, созданные компонентами, необходимо передать другим компонентам. В системах, где целевой компонент у запроса может быть один или запросы распределяются по схеме, распределяющей нагрузку, маршрутизация запросов – тривиальная задача. Она может быть решена статическими таблицами, сопоставляющими типы запросов к компонентам в сочетании с

алгоритмом round-robin (поочерёдное распределение задач). Но в данном случае задача усложняется. Выбор целевого компонента запроса может зависеть от нескольких параметров. Параметры могут быть статистическими (количество активных запросов в данный момент, процент успешным запросов, частота запросов и т. д.), внешними (информация, которую необходимо запросить у компонентов в момент запроса) или заданными статически. Соответственно, алгоритмы, принимающие решение о конечной точке запроса, в теории могут быть сложными и массивными.

Определение конечной точки запросов не единственная задача, которую необходимо решать. В подсистеме необходимы активные модули, которые выполняют задачи в зависимости от событий или в определённые промежутки времени, как репликация данных между хранилищами значений переменных для повышения отказоустойчивости. Необходима поддержка запросов, которые не могут быть направлены ни в один компонент и касаются системы в целом, как запрос на получение показателей качества по компонентам. Очевидно, что решать все перечисленные задачи таблицами или алгоритмами внутри самой подсистемы интеграции будет ошибкой, поскольку это лишит её гибкости и усложнит развитие всей системы активных знаний.

Чтобы позволить системе развиваться, масштабироваться и улучшать логику работы необходимо предоставить возможности и необходимую информацию управляющим модулям, которые смогут забрать на себя часть ответственности за общее функционирование системы активных знаний. Назовём такие модули операторами. Это программное обеспечение, работающее отдельно от подсистемы интеграции, но активно с ним взаимодействующие. Для коммуникации между ними необходим свой способ общения. Формат клиент-сервер в данном случае будет не подходящим, поскольку они равнозначны друг для друга. Лучше всего здесь подойдёт формат взаимодействия, основанный на событиях. События – это сущности, генерируемые как подсистемой интеграции, так и операторами. Со стороны первой события в основном связаны с жизненным циклом запроса: запрос получен, разрешён, отправлен, окончен. В начале своего

жизненного цикла операторы могут, но не обязаны, подписаться на получения необходимых им событий. Если оператор хочет взять на себя ответственность за разрешение запросов – он должен подписаться на событие получение запроса.

Для обеспечения эффективности передачи данных в разработанном решении операторы не имеют доступа к телу запроса и работают только с именем запроса, его видом, параметрами и прочим. Это обеспечивает как целостность данных (поток данных контролируется только подсистемой интеграции), так и эффективность. Тело запроса не должно помещаться в буфер или передаваться узлам-посредникам, поскольку это приведёт к задержкам и переполнению дискового пространства или оперативной памяти. Но в некоторых случаях это может привести к ограничениям, которые невозможно решить в рамках одного запроса. Эти ограничения связаны с невозможностью направить данные нескольким узлам поочерёдно. Требуются доработки, но в данной работе эти ограничения будут приняты как издержки разработанной архитектуры.

Операторы не только слушают, но и генерируют события. В них входят события-ответы на события подсистемы интеграции (конечная точка запроса или ответ на запрос) и запросы от самого оператора с возможностью отправить его конкретному компоненту.

Для того, чтобы у операторов был достаточный контекст в теле события также передаётся список всех компонентов. Список событий, передаваемых между подсистемой и операторами представлен ниже:

1. newRequest
   - Описание: события, связанные с обращением компонента к подсистеме интеграции с новым запросом;
   - Тело сообщения:
     - name: “имя запроса”,
     - requestType: “GET” или “SET”,
     - parameters: {параметры запроса},
     - components: [{id: …, group: “группа компонента”}, …],
     - correlationId: идентификатор сообщения для ответа

2. requestFinished
   - Описание: событие завершения запроса;
   - Тело сообщения:
     - name: “имя запроса”,
     - requestType: “GET” или “SET”,
     - parameters: {параметры запроса},
     - components: [{id: …, group: “группа компонента”}, …],
     - requestResult: ничего если requestType=get или ошибка, иначе результат запроса
     - requestError: ничего если запрос завершён успешно, иначе ошибка
     - correlationId: идентификатор сообщения для ответа
3. newRequestTarget
   - Описание: ответ от оператора с обозначением выбранной конечной точки запроса;
   - Тело сообщения:
     - correlationId: идентификатор сообщения для ответа,
     - componentId: идентификатор компонента
4. newRequestResponse
   - Описание: ответ от оператора с результатом запроса;
   - Тело сообщения:
     - correlationId: идентификатор сообщения для ответа,
     - buffer: результат запроса в виде массива байт
5. subscribe
   - Описание: Подписка оператора на получение событий от подсистемы интеграции;
   - Тело сообщения:
     - subscribes: массив фильтров, частичных значений тел событий (в том числе название события), которые определяют отправляемые события.

6. makeRequest

- Описание: сделать запрос к компоненту от оператора;
- Тело сообщения:
  - componentId: идентификатор компонента (не обязателен),
  - name: “имя запроса”,
  - correlationId: идентификатор сообщения для ответа,
  - parameters: {параметры запроса}

7. makeRequestResponse

- Описание: ответ от подсистемы интеграции на запрос оператора;
- Тело сообщения:
  - correlationId: идентификатор сообщения для ответа,
  - buffer: результат запроса в виде массива байт

8. makeRetranslation

- Описание: отправить get запрос к одному компоненту и переслать ответ другому в set запросе. Оператору придёт результат set запроса в makeRequestResponse;
- Тело сообщения:
  - getRequest: формат из makeRequest,
  - setRequest: формат из makeRequest,
  - correlationId: идентификатор сообщения для ответа

3.4 Архитектура подсистемы интеграции

Предложена архитектура, состоящая из следующих компонентов:

- Servers (Серверы): принимает и обрабатывает запросы и сообщения к подсистеме интеграции;
- RequestManager (Управляющий запросами): обрабатывает запросы и проводит маршрутизацию;
- Endpoints (Конечные точки): управляет конечными точками запросов;
- PipeBuilder (Конструктор потока данных): проводит потоковую передачу данных между узлами системы;

– EventBus (Шина событий): передаёт события между компонентами подсистемы интеграции.

Каждый компонент архитектуры был введён для решения поставленных задач и разрабатывался с упором на масштабируемость и гибкость. Диаграмма классов в упрощённом виде с основными элементами представлена на рисунке 2, диаграмма компонентов на рисунке 3. Рассмотрим каждый компонент подробнее.

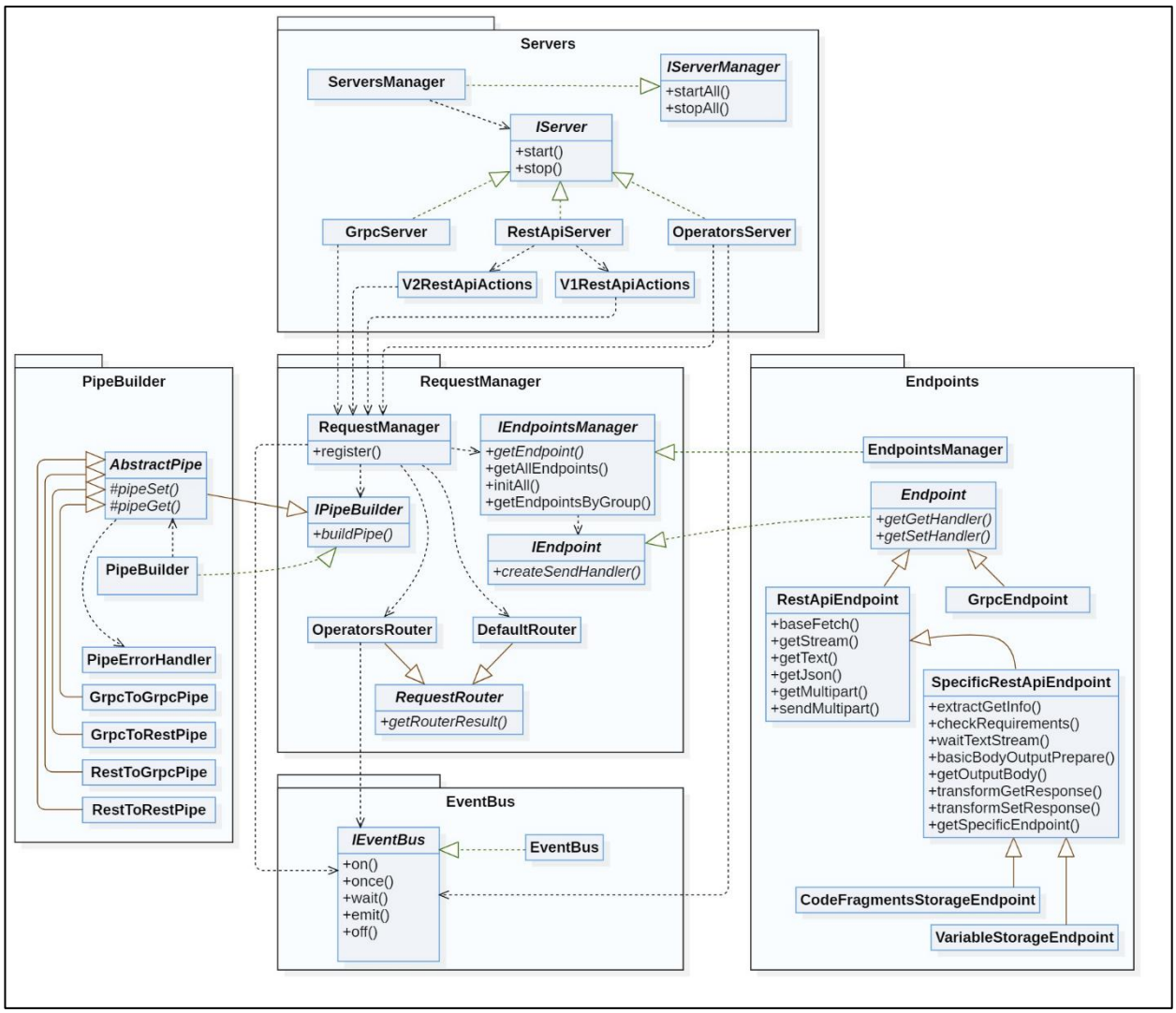

Рисунок 2 — Диаграмма классов подсистемы интеграции

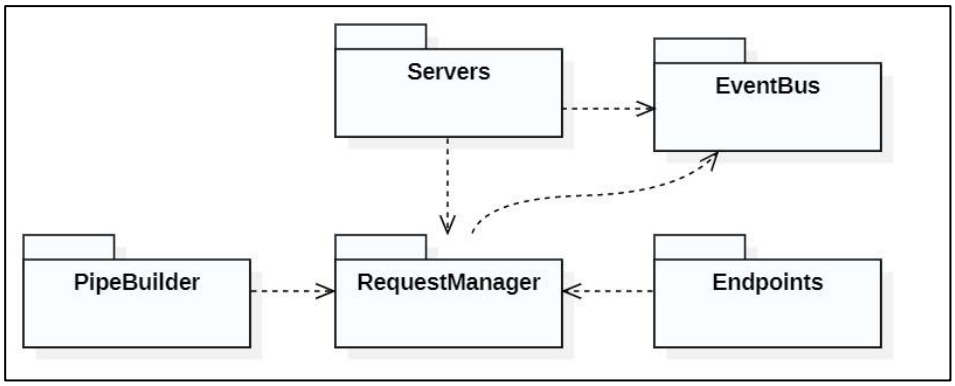

Рисунок 3 — Диаграмма компонентов подсистемы интеграции

3.3.1 Компонент Servers

В компоненте Servers запускаются серверы, принимающие запросы и сообщения извне в подсистему интеграции. Для управления серверами используется класс ServersManager, который запускает и останавливает их. Количество серверов зависит от требуемых протоколов передачи данных, число которых не ограничено и может увеличиваться в дальнейшем, обеспечивая масштабируемость. В архитектуру заложено 3 сервера: GrpcServer (сервер gRPC протокола), RestApiServer (HTTP RestAPI сервер) и OperatorsServer (сервер, обрабатывающий события от операторов). Все серверы направляют запросы в обработанном виде в класс RequestManager для его регистрации и дальнейшей обработки. OperatorsServer, помимо регистрации запросов, также регистрирует события, отправленные операторами, в EventBus для дальнейшего перехватывания и использования другими операторами или в OperatorsRouter.

3.3.2 Компонент RequestManager

RequestManager является центральным в обработке запросов и функционировании подсистемы интеграции в целом. Он осуществляет весь контроль над зарегистрированным запросом, а именно выполняет маршрутизацию, получает дескриптор конечной точки запроса используя EndpointsManager, открывает поток передачи данных между отправителем и получателем запроса с помощью PipeBuilder, а также регистрирует все события жизненного цикла запроса используя EventBus. Стоит отметить, что на компонент RequestManger возложена наибольшая ответственность, поэтому

применён принцип инвертирования зависимостей, благодаря чему все компоненты, кроме EventBus, зависят от RequestManager, а поток данных в случае PipeBuilder и Endpoints противоположный.

Маршрутизация проходит в OperatorsRouter и DefaultRouter. Первый направляет запрос ответственному оператору. Отправка осуществляется с помощью EventBus и далее OperatorsServer, как и ожидание ответного события от оператора. Если ответственного оператора нет (он не подписался на событие получения запроса) используется базовый DefaultRouter со стандартными определёнными конечными точками запросов.

3.3.3 Компонент Endpoints

Каждый компонент в системе может реализовать интерфейс удобным для себя образом. Он также может быть разработан для другой системы со своими форматами взаимодействия, но функционально быть полезным для применения. Для поддержки узлов с неоднородными интерфейсам разработан компонент Endpoints, в котором определяются адаптеры, выполняющие трансформацию запросов из внутреннего формата запросов подсистемы в требуемый для компонента и наоборот. Для быстрой и эффективной поддержки используются базовые классы RestApiEndpoint+SpecificRestApiEndpoint и GrpcEndpoint, в которых содержатся вспомогательные функции для адаптации. Управляет адаптерами класс EndpointsManager, который предоставляет доступ к конечным точкам и отслеживает их актуальность.

3.3.4 Компонент PipeBuilder

Запрос можно разделить на две части: дескриптор запроса, полученный от одного из серверов и другой дескриптор, полученный от адаптера конечной точки запроса. Их необходимо связать между собой, образовав поток передачи данных между узлами. Для этого введён компонент PipeBuilder, конструирующий и контролирующий поток данных, при этом преобразуя его при передаче данных между компонентами с разными протоколами (gRPC к HTTP и наоборот).

3.3.5 Компонент EventBus

События подсистемы интеграции необходимо передавать между её частями и для этого определена шины событий, класс EventBus. Позволяет генерировать события, подписываться на них и ожидать ответного события.

3.3.6 Поток данных

Поток данных, как и последовательность работы подсистемы интеграции в целом, можно описать следующим образом:

1. Запросы от узлов системы или сообщения от оператора поступают в компонент Servers, операторы подписываются на оповещение о событиях, необходимых им для работы;
2. Запрос с дескриптором от сервера регистрируется в компоненте RequestManager, генерируется событие начала запроса в EventBus, в него добавляются данные из EndpointsManager;
3. Операторам, подписанным на событие начала запросов, отправляются сообщения;
4. RequestManager пытается осуществить маршрутизацию в OperatorsRouter, который ожидает события разрешения запроса от операторов;
5. Получив ошибку или отказ RequestManager проводит маршрутизацию в DefaultRouter;
6. После маршрутизации, если результатом стал ответ на запрос, он возвращается запросившему узлу и запрос завершается, иначе в EndpointsManager запрашивается дескриптор определённой конечной точки запроса;
7. EndpointsManager использует адаптер запрошенного компонента и создаёт дескриптор запроса для конечной точки запроса;
8. RequestManager передаёт оба дескриптора в PipeBuilder, который конструирует поток передачи данных. После закрытия потока запрос считается завершённым.

3.3.7 Пример запроса

Для примера возьмём запрос на получение значения переменной, отправленный пользовательским интерфейсом:

1. При запуске подсистемы интеграции операторы подписываются на получение необходимых им событий путём отправки соответствующего сообщения события. Сообщения с событиями от операторов обрабатывается в классе OperatorsServer компонента Servers. Оператор хранилищ значений переменных подписывается на события начала запросов, связанных со значениями переменных, в частности получения и установки значения переменной, тем самым принимает на себя ответственность за их разрешение;

2. Запрос поступает в сервер в классе RestApiServer компонента Servers, он преобразуется во внутренний формат подсистемы интеграции и обработчики данных запроса/ответа, формируя дескриптор запроса, и передаётся в класс RequestManager одноимённого компонента;

3. RequestManager отправляет событие начала запроса в класс EventBus. OperatorsServer слушает все события из EventBus и направляет их операторам в соответствии с подписками. RequestManager пытается узнать куда передать запрос от операторов в классе OperatorsRouter;

4. OperatorsRouter ожидает события разрешённой конечной точки запроса или ответа на запрос от EventBus, куда оно попадёт от OperatorsServer и ответственного оператора;

5. Оператор хранилищ значений переменных, получив событие начала запроса и вычислив конечную точку по некоторому алгоритму, возвращает её путём отправки сообщения события разрешённой конечной точки запроса, которой является идентификатор одного из компонентов системы;

6. RequestManager после получение идентификатора компонента запрашивает у класса EndpointsManager дескриптор конечной точки запроса. Он обращается к адаптеру требуемого компонента (хранилища значений переменных) и получает обработчики данных запроса/ответа;

7. RequestManager, получив оба дескриптора направляет их в класс PipeBuilder одноимённого компонента, чтобы построить двунаправленный поток данных между обработчиками данных запросов и ответов.

3.5 Разработка алгоритмов операторов

Для системы LuNA, помимо самой подсистемы интеграции, также необходимо разработать операторы, поддерживающие общую логику систему. Конечно, разработать все требуемые операторы невозможно, поскольку задача, которую они решают – поддержка масштабируемости и гибкости системы, что подразумевает постоянное их добавление и изменение в зависимости от поступающих требований. Поэтому в рамках данной работы будут описаны операторы, демонстрирующие концепцию и потенциал для дальнейших доработок, но при этом необходимые в системе.

3.5.1 Оператор хранилищ значений переменных

В системе может быть множество хранилищ значений переменных. Они обрабатывают запросы на чтение, запись, удаление и получение списка значений переменных. Но при обращении компонентов к подсистеме интеграции с описанными запросами возникает проблема – конечная точка запроса не определена. Более того, запрос на получение списка значений переменных, например, требует агрегации информации от всех хранилищ переменных. Нужен механизм обработки запросов, который будет контролировать и направлять их в нужные хранилища – оператор хранилищ значений переменных. Также можно реализовать распределённое хранение с разделением данных значений переменных между разными хранилищами, однако в рамках данной работы нет смысла разрабатывать подобные алгоритмы. Ограничимся необходимым минимумом – направлять запросы и обеспечивать работоспособность системы. Разберём каждый запрос и алгоритм его работы:

1. Добавление значения переменной:

– Оператор получает список всех компонентов вместе с событием начала запроса;

– Необходимо запросить доступное пространство в каждом хранилище, но на данный момент у компонентов такой запрос не реализован, так что допустим, что у хранилищ доступное пространство не ограничено. Тогда оператору достаточно направлять запросы каждому хранилищу поочерёдно. Представленные алгоритмы, как и операторы, примерные и могут модернизироваться и улучшаться в любой момент при изменении и доработке системы;

2. Получение списка значений переменных:

– Оператор запрашивает у всех хранилищ переменных их списки. Очевидно, что в долгосрочной перспективе хранилищ значений переменных, как и самих значений, может быть достаточно, чтобы сделать данный запрос крайне неэффективным из-за расходов на запросы всем компонентам и отправку ответа. Но поскольку алгоритмы примерные и не несут ценности сами по себе, примем число хранилищ не более 10, а количество значений не более 1000;

– Объединив и убрав повторы, возвращает готовый список в качестве ответа на запрос;

– При каждом запросе на получение списка значений проводится кэширование, в каких компонентах какие значения переменных имеются, для дальнейшего использования.

3. Получение значения переменной:

– Оператор пытается найти в кэше запрошенный идентификатор. Если он найден, а хранилище с ним связанное всё ещё подключено к подсистеме интеграции – направляет запрос в него;

– Иначе запрашивается список всех значений переменных в каждом хранилище и ищет в них запрошенный идентификатор;

– Если компонент не был найден – вернуть в качестве результата запроса ошибку.

4. Удаление значения переменной:

– Оператор отправляет всем хранилищам переменных запрос на удаление значения переменной с указанным

в запросе идентификатором, возвращает пустой успешный результат после завершения.

3.5.2 Оператор репликации значений переменных

Для повышения отказоустойчивости полезно копировать сохранённое значение переменной в другие хранилища. Так при выходе из строя одного из них может быть использовано другое. Для этого можно реализовать оператор репликатор. Опишем алгоритм его работы:

– Оператор слушает только одно событие – успешное завершение запроса добавления значения переменной, в которое будет включён идентификатор новой переменной;

– Направляет запросы на создание потока между исходным и несколькими (например, двумя) случайным хранилищам: запрос получения значения переменной -> запрос на добавление значения переменной. Так, значение будет храниться сразу в нескольких хранилищах. Механизм репликации может быть и сложнее, учитывать доступное пространство и «надёжность» компонентов;

– Оператор хранилищ значений переменных, получив событие об успешном завершении запросов, связывает новые хранилища с идентификатором значения для дальнейшего использования.

3.5.3 Оператор контроля качества

Операторы направляют запросы в основном опираясь на самостоятельно получаемые сведения, как например оператор хранилищ значений переменных опрашивает, в каком хранилище есть искомое значение. Но также помимо прямых сведений для принятия решения может понадобиться косвенная информация. Допустим, одно и то же значение переменной есть в двух хранилищах, но одно из них по какой-то причине часто отключается или возвращает ошибки. Тогда можно сделать вывод, что лучше передать запрос более стабильному компоненту. Таких косвенных параметров может быть неограниченное число: количество активных запросов к компоненту или от него, процент успешных или провальных ответов по каждому виду запросов, запросы

в минуту, процент времени вне сети и прочее. Эти сведения могут понадобиться многим операторам и компонентам, так что нет смысла вычислять их в каждом. Более эффективным решением будет создать оператор контроля качества. В его задачи входит собирать все события в системе, проверять доступность компонентов, вычислять статистические сведения и предоставлять их по запросу получения показателей качества. Соответственно, у него будет своё системное имя запроса. Опишем базовый алгоритм его работы:

– Оператор слушает события: начала всех запросов, окончания всех запросов, включая успешные и провальные, начало запроса получения показателей качества;

– При запросе получения показателей качества формирует ответ для каждого компонента, который участвовал в выполнении запросов (т. е. упоминался в событиях): количество активных запросов к компоненту и от компонента, количества провальных, успешных и сумма запросов, отправленных компоненту, процент успешных и провальных запросов к компоненту, число запросов к компоненту в час. Характеристики могут дополняться и изменяться при необходимости;

– Другие операторы могут запросить показатели качества у подсистемы интеграции, которые будут получены у оператора контроля качества;

3.5.4 Оператор исполнителей

Как и в случае с хранилищами значений переменных, исполнителей может быть несколько, но выбор исполнителя для запроса связан с более сложной логикой, поскольку необходимо как контролировать нагрузку на них, так и учитывать качество компонентов, чтобы сохранить баланс между скоростью исполнения и отказоустойчивостью. Если у исполнителя, например, провальных запросов больше 70%, то имеет смысл давать больше нагрузки более стабильным исполнителям, хоть и с потерей скорости исполнения. Сведения о качестве компонентов получим в операторе контроля качества.

Также на выбор исполнителя влияет содержание запроса, а именно требуемое оборудование, наличие которого обязательно. Определить, может ли

исполнитель взять на себя этот запрос, можно обращениями к ним с содержанием «можешь ли ты исполнить это?». Другой подход – использовать «аукцион», в котором каждый исполнитель указывал бы, насколько данный запрос подходит под его оборудование и окружение. Также у запроса могут быть метапараметры на исполнение с максимальной скоростью или гарантией выполнения запроса, что повлияет на выбор в соответствии со статистикой. Алгоритм выбора исполнителя может изменяться вместе с модернизацией системы, но опишем его в соответствии с возможностями, доступными на данный момент:

– Оператор слушает события начала запрос на исполнение программы или фрагмента кода;

– Запрашивает у подсистемы интеграции и далее оператора контроля качества сводку характеристик по каждому компоненту, выбирает из неё компоненты исполнителей, подключённых к подсистеме интеграции;

– Вычисляет коэффициент для каждого компонента, опираясь на нагрузку и процент провальных запросов, и выбирает компонент с максимальным коэффициентом.

4 РЕАЛИЗАЦИЯ ПОДСИСТЕМЫ ИНТЕГРАЦИИ

4.1 Выбор средств и технологий реализации

Для реализации подсистемы интеграции использовалась платформа Node.Js [13] и язык JavaScript с надстройкой TypeScript [14]. Это решение обусловлено тем, что JavaScript и, в частности, Node.Js хорошо подходит для обработки множества одновременных запросов и поддерживает событийную модель программирования. Основная проблема, с которой может столкнуться JavaScript – тяжёлые синхронные (алгоритмические) вычисления, из-за которых цикл исполнения заблокируется. Но в данном случае такие вычисления отсутствуют по определению и переносятся на сторону операторов.

Связь между компонентами и подсистемой интеграции осуществляется по протоколам HTTP/1.1 и gRPC (HTTP/2) [1, 4, 6]. Протокол HTTP/1.1 использован, как стандартный и широко используемый, все узлы в системе обмениваются сообщениями используя его. Также пользовательский интерфейс может обращаться к подсистеме интеграции только с его помощью. Второй протокол, gRPC, поддержан как более эффективный и надёжный, но он требует миграции в дальнейшем и в данный момент не используется. Для связи по HTTP используется библиотека Fastify [15], по gRPC одноимённая библиотека для Node.Js [16].

Операторы, как и подсистема интеграции, реализованы на Node.Js, хотя способ реализации их, как и компонентов, может сильно отличаться. Для связи используется WebSocket и библиотека Socket.IO [17], которая позволит организовать двунаправленный обмен сообщениями с событиями.

Для управления изменениями исходного кода использовалась системы контроля версий Git [18]. Исходный код хранится на платформе GitHub [19].

4.2 Реализация форматов передачи данных

4.2.1 Запросы gRPC

Формат запросов gRPC описан в proto файлах, которые полностью декларируют все типы запросов и сообщений. По proto файлам генерируется код для любого популярного языка программирования, что поддерживает язык-

независимость и простоту построения интерфейсов, но в данном случае генерируется код для TypeScript. Структуры, полученные таким образом, послужат основой для внутреннего формата запросов подсистемы интеграции.

В соответствии с форматом, описанным в разделе 3.2 необходимо реализовать два запроса, Get и Set. На Листинге 1 приведены proto файлы, описывающие их. Rpc (удалённый вызов процедуры) Get принимает на вход GetInfo, информационный объект, описывающий предмет запроса. Он состоит из RequestType (тип запроса) и объекта с необходимой для запроса информацией, например, для значения переменной – BasicIdGet с одним полем id, идентификатором. Результатом запроса станет поток из сообщений DataStream, которые могут быть либо информационным объектом DataInfo в первом пакете, либо фрагментом данных bytes во всех последующих. DataInfo содержит также RequestType, тип данных DataType и информационный объект. DataType определяет, как будет передана информация и в каком виде, но на данный момент используется только BYTES, массив байт.

Запрос Set, как можно заметить, является инвертированным Get запросом, то есть принимает на вход поток DataStream, а возвращает GetInfo. Отличие только в том, что в DataStream.DataInfo можно отправить информационный объект, описывающий данные, как при запросе на сохранение фрагмента кода можно приложить BasicGetInfoData с идентификатором, чтобы обновить его.

RequestType это тип запроса. Он назначен для каждого запроса, но один и тот же тип может использоваться и в Set, и в Get запросах, как VAR_VALUE (значение переменной) используется для сохранения и получения значения.

Таким образом, используя структуры, написанные в proto файлах, мы можем описать любой запрос в системе. Для поддержки обратной совместимости при доработках все номера полей имеют запас для добавления и удаления, что не повредит интерфейсы, функционирующие на старых версиях типов.

```
service MainRequests {
  rpc Get(GetInfo) returns (stream DataStream) {}
  rpc Set(stream DataStream) returns (GetInfo) {}
}

message DataInfo {
  RequestType request_type = 1;
  DataType data_type = 2;
  oneof data_value_type {
    BasicGetInfoData code_f = 50;
    Empty code_f_info = 52;
    Empty code_f_list = 54;
    Empty code_f_plugins_list = 56;
    Empty code_f_plugin_procedure = 58;
    BasicGetInfoData code_f_plugin = 60;

    Empty var_value = 102;
    Empty var_value_list = 104;
    BasicGetInfoData var_value_delete = 106;
    BasicGetInfoData var_value_meta = 108;
    BasicGetInfoData var_value_meta_delete = 110;

    BasicSettableJsonData custom = 501;
  }
}

message BasicIdGet {
  string id = 1;
}

message CodeFPluginProcedureGet {
  string code_f_id = 1;
  string type = 2;
}

enum DataType {
  UNKNOWN_DATA_TYPE = 0;
  NONE = 10;
  TEXT = 20;
  JSON = 30;
  LINK = 40;
  BYTES = 50;
}
```

```
message DataStream {
  oneof info_or_data {
    DataInfo info = 3;
    bytes chunk_data = 10;
  }
}

message GetInfo {
  RequestType request_type = 1;
  oneof info_type {
    BasicIdGet code_f_get = 52;
    BasicIdGet code_f_info_get = 54;
    BasicIdGet code_f_plugin_get = 56;
    CodeFPluginProcedureGet
code_f_plugin_procedure_get = 58;
    BasicIdGet code_f_plugins_list_get = 60;

    BasicIdGet var_value_get = 102;
    BasicIdGet var_value_meta_get = 104;


    string response = 500;
    BasicIdGet custom = 501;
  };
}

message BasicGetInfoData {
  BasicIdGet getInfo = 1;
}

message Empty {}

enum RequestType {
  UNKNOWN_REQUEST_TYPE = 0;
  OP_QUALITY_REPORT = 100;

  CODE_F = 202;
  CODE_F_LIST = 204;
  CODE_F_INFO = 206;
  CODE_F_PLUGIN = 208;
  CODE_F_PLUGINS_LIST = 210;
  CODE_F_PLUGIN_PROCEDURE = 212;

  VAR_VALUE = 302;
  VAR_VALUE_DELETE = 304;
  VAR_VALUE_LIST = 306;
  VAR_VALUE_META = 308;
  VAR_VALUE_META_DELETE = 310;
}
```

Листинг 1 — Proto файлы gRPC запросов

4.2.2 Запросы HTTP

Интерфейс запросов, передаваемых по HTTP, выполнен в двух вариантах. Первый в основном дублирует интерфейс запросов gRPC, второй более удобный для использования клиентами и соответствующий стандарту. Они

разрабатывались последовательно и соответствуют первой и второй версии интерфейса.

В первой версии HTTP интерфейса запросам Get и Set соответствуют POST /api/v1/get и POST /api/v1/set. В /api/v1/get предмет запроса GetInfo отправляется в application/json теле запроса. В ответ возвращается multipart/form-data с двумя полями: info с application/json DataInfo и data с массивом байт. В /api/v1/set наоборот отправляется multipart/form-data с DataInfo в поле info и массив байт в поле data. На рисунке 4 представлены примеры запросов на получение и сохранение значений переменных.

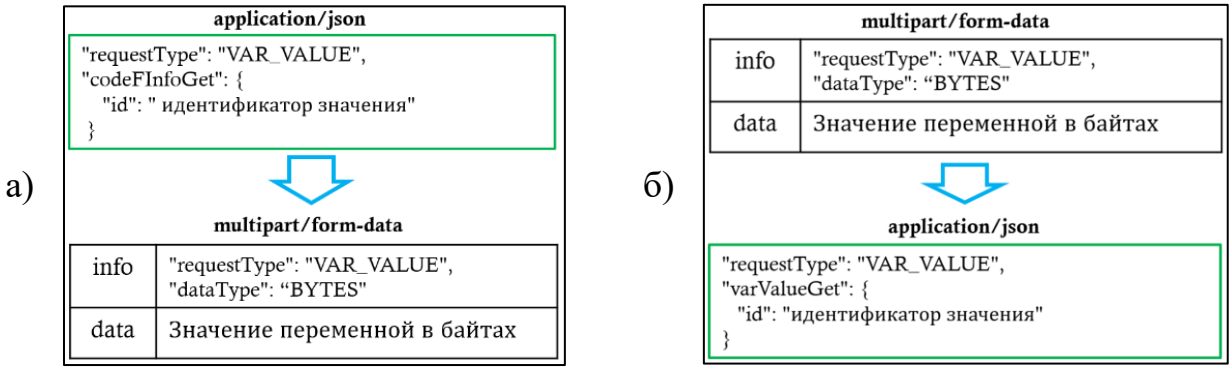

Рисунок 4 — Пример get и set HTTP запросов

Вторая версия в отличии от первой для каждого запроса задаёт свой URI. Информационных объектов больше нет, вся необходимая информация передаётся в пути и параметрах HTTP запроса. Общий вид запросов: GET|POST /api/v2/ [группа запросов] / [предмет запроса] / [идентификатор]? [прочие параметры запроса]. Данные передаются также в multipart/form-data, но с одним полем, data. С одной стороны, вместо multipart/form-data можно было бы использовать просто octet-stream, но так есть возможность для расширения, добавления дополнительной метаинформации или опций в дальнейшем. Все запросы приведены в таблице 2.

Таблица 2 – Запросы второй версии HTTP интерфейса

Описание	**Запрос**	**Тело запроса**	**Ответ**
Запрос списка фрагментов кода	GET /api/v2/code-f-storage /list		200 **Multipart: data** octet-stream: json список

Описание	Запрос	Тело запроса	Ответ
Запрос информации о фрагменте кода	GET /api/v2/code-f-storage /info/{id}		200 **Multipart:** **data** octet-stream json описание
Добавить фрагмент кода	POST /api/v2/code-f-storage /fragment/{id}	**Multipart:** **data** octet-stream архив tar	204 *No content*
Получить файлы фрагмента кода	GET /api/v2/code-f-storage /fragment/{id}		200 **Multipart:** **data** octet-stream архив tar
Получить обработанный плагином фрагмент кода	GET /api/v2/code-f-storage /procedure/{id}?type="плагин"		200 **Multipart:** **data** octet-stream json с фрагментом кода
Получить список плагинов ФК	GET /api/v2/code-f-storage /plugins-list		200 **Multipart:** **data** octet-stream json список
Получить плагин	GET /api/v2/code-f-storage /plugin/{id}		200 **Multipart:** **data** octet-stream архив tar
Добавить плагин	POST /api/v2/code-f-storage /plugin/{id}	**Multipart:** **data** octet-stream архив tar	204 *No content*
Получить список значений переменных	GET /api/v2/var-storage /list		200 **Multipart:** **data** octet-stream json список
Получить значение переменной	GET /api/v2/var-storage /value/{id}		200 **Multipart:** **data** octet-stream массив байт
Добавить значение переменной	POST /api/v2/var-storage /value	**Multipart:** **data** octet-stream массив байт	201 application/json {id: string}
Удалить значение переменной	DELETE /api/v2/var-storage /value/{id}		204 *No content*

4.3 Реализация адаптеров

Для обращения к компонентам от подсистемы интеграции необходимо реализовать адаптеры, преобразовывающие внутренний формат запросов в требуемый интерфейс и инициировать запрос. Для этого необходим функционал для настройки под различные компоненты со своей спецификой, который реализован в абстрактном классе SpecificRestApiEndpoint, от которого

наследуются и который реализуют специфичные REST API адаптеры. На Листинге 2 приведён пример адаптации запроса на получение значения переменной в VariableStorageEndpoint. Как видно, чтобы построить адаптер интерфейса достаточно описать объект конфигурации, что ускоряет адаптацию и изменение интерфейсов компонентов, все остальные действия выполнит getSpecificEndpoint и прочие вспомогательные методы.

```
const valstPrefix = "/valst";

const uris = {
  getValue: ["GET", (id: string) =>
    `${valstPrefix}/${id}`],
  ...
}

protected getValue(info: GetInfo_Strict) {
  const uri = uris.getValue;
  return this.getSpecificEndpoint(info, {
    type: "GET",
    getInfoName: "varValueGet",
    requirements: ["id"],
    inputOptions: {
      uri: (info) => uri[1](info!.id!),
      httpMethod: uri[0]
   }
  })
}
```

Листинг 2 — Фрагмент VariableStorageEndpoint

В рамках данной работы в соответствии с описанным разработанным интерфейсом реализованы адаптеры для нескольких компонентов системы: хранилище значений переменных (VariableStorageEndpoint) и библиотека фрагментов кода (CodeFragmentsEndpoint). Разработка компонентов для других компонентов потребует описать конфигурацию для запросов и дополнить proto файлы.

4.4 Реализация операторов для ядра

Для реализации оператора достаточно написать программу с использованием любых средств, которая бы подключалась к подсистеме интеграции по WebSocket и обменивалась с ним событиями. Сам процесс создания операторов сильно упрощается, поскольку необходимо только написать обработчики событий, которые могут быть любой сложности.

Был разработан оператор хранилищ значений переменных, алгоритм которого описан в разделе 3.4. Для примера рассмотрим фрагмент этого оператора, обрабатывающий запрос на получение списка значений переменных на Листинге 3. Для каждого хранилища вызывается requestValueList с отправкой события makeRequest запроса на получение списка значений к определённому компоненту, ожидается ответное событие makeRequestRepsonse с результатом запроса. Затем все списки объединяются в один и отправляются как событие newRequestResponse с ответом на запрос. По итогу события скрывают от операторов логику работы подсистемы интеграции и её форматы данных. Это позволит разработчикам сконцентрировать внимание на алгоритмах вычисления ответа, а не на проблемах соединения и форматов запросов.

```
if (event.type === "VAR_VALUE_LIST") {
  const lists = await Promise.all(storages.map(async (component) => {
    return await requestValuesList(component.id) || [];
  }));
  const flatList = Array.from(new Set(lists.flat()));
  return eventBus.emit({
    eventName: "new-request-response",
    buffer: Buffer.from(JSON.stringify(flatList))
  }, {
    correlationId: corrId
  });
}

async function requestValuesList(componentId: string) {
  const listCorrId = eventBus.emit({
    eventName: "make-request",
    type: "VAR_VALUE_LIST",
    name: "GET",
    componentId
  });
  const listEvent = await eventBus.wait(listCorrId, 5000);
  if (listEvent.eventName === "make-request-response" && listEvent.buffer) {
    cache.forEach((value, key) => {
      if (value === componentId) {
        cache.delete(key);
      }
    });
    const list = JSON.parse(listEvent.buffer.toString()) as string[];
    list.forEach(id => {
      cache.set(id, componentId);
    })
    return list;
  }
}
```

Листинг 3 — Фрагмент VariableStorageOperator

5 ТЕСТРОВАНИЕ

5.1 Описание сценариев тестирования

Для тестирования системы необходимо провести интеграционное тестирование всей системы и тест производительности.

Сценарий интеграционного тестирования:

– Участники: Подсистема интеграции; UI; Хранилище значений переменных 1, 2, 3; Оператор хранилищ значений переменных;

– Сценарий работы:

1. UI запрашивает создание значения переменной 5 раз.
2. Подсистема интеграции принимает запрос, отправляет сообщение с событием оператору;
3. Оператор определяет хранилище значений переменных для сохранения;
4. После сохранения значения отправляется событие о завершении запроса с идентификатором нового значения, которое кэшируется в операторе. Далее алгоритм передачи запроса будут опущен;
5. UI запрашивает список доступных переменных и выводит его;
6. UI запрашивает значение значения по идентификаторам из списка и выводит их;
7. UI запрашивает и ожидает удаление каждого значения переменной;
8. UI запрашивает список доступных переменных и выводит его.

Для тестирования производительности проведём стресс-тест для определения пропускной способности подсистемы интеграции: запустим 2000 запросов на добавление значения переменной в течение 2 секунд и определим число выполненных в секунду запросов, максимальное число активных запросов и максимальное время выполнения запроса.

5.2 Подготовка тестирования

Для тестирования операторов и подсистемы интеграции необходима среда, в которой можно контролировать число компонентов, их наполнение, свойства и интерфейс без вреда для основной инстанции системы. Для этого разработана

тестовая установка, в которой можно быстро запускать подсистему интеграции, компоненты, операторы и имитацию пользовательского интерфейса, которая является программой, выполняющий запросы к подсистеме интеграции. Каждая часть по отдельности пишет журналы, которые объединяются в один файл. Файловая структура представлена на рисунке 5. В ней scenario-launcher – точка входа, которая запускает все другие части, в папке src находится исходный код подсистемы интеграции, test-operators содержит и запускает операторы, test-endpoints компоненты, test-ui – имитация пользовательского интерфейса. Как можно заметить, в каждой папке, кроме scenario-runner, имеется файл «pid.txt». Он хранит PID процесса и используется, чтобы гарантировать завершение предыдущей запущенной инстанции при запуске новой. В папках logs находятся журналы с выводимыми сообщениями, что поможет при проверке работы сценариев. Код пользовательского интерфейса, используемого в сценарии, приведён в Приложении А.

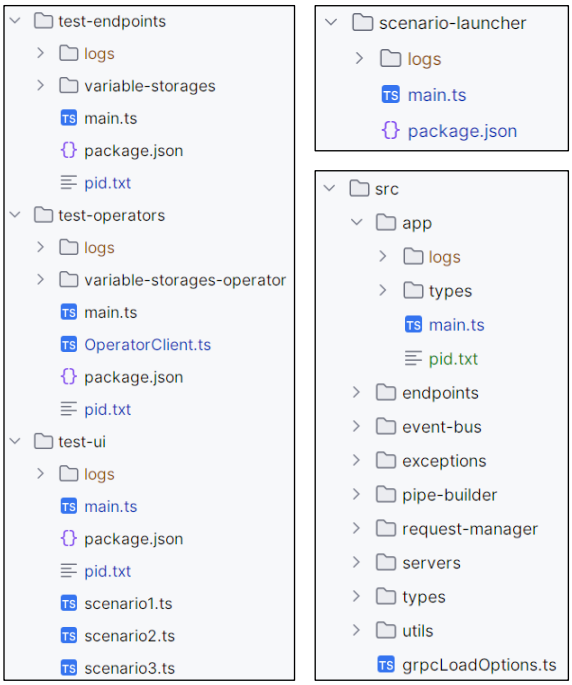

Рисунок 5 — Файловая система тестовой установки

Для стресс-теста используем программу Apache JMeter [20], которая позволяет гибко настроить сценарии выполнения запросов, выполнить их определённое число раз за временной интервал и записать все результаты в файлы в требуемом формате. Структура теста представлена на рисунке 6.

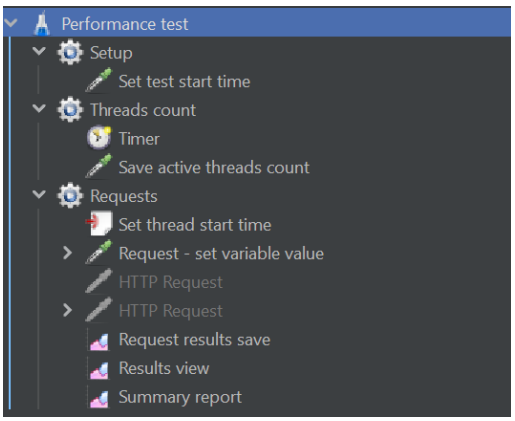

Рисунок 6 — Структура стресс-теста в JMeter

5.3 Результаты тестирования

В результате интеграционного тестирования был получен журнал работы системы, частично представленный в Приложении Б. Каждое сообщение маркировано временной меткой, источником записи (CORE, UI, OPERATORS и ENDPOINTS), уточнённым источником (variablesStorage1,2,3 и др.). Как видно, в конце, после удаления каждого значения, был выведен пустой массив списка значений переменных.

Результаты стресс теста в виде характеристик представлены на Рисунке 7. Samples это количество запросов, Min, Max, Average и Std. Dev. характеристики времени исполнения запросов, Throughput пропускная способность. График теста представлен на рисунке 8. Красные точки на нём – записи завершённых запросов, на оси X время их начала, на оси Y время исполнения. Синяя линяя это число активных запросов в момент времени. По итогу можно сказать, что подсистема интеграции успешно справляется с поступающим потоком сообщений. Пропускная способность 1000 запросов в секунду, максимальное количество одновременно активных запросов 340, максимальное время выполнения 66 мс, ошибки отсутствуют.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB...	Sent KB/sec	Avg. Bytes
Request - set variable value	2000	12	0	66	17.02	0.00%	1000.0/sec	202.50	487.20	207.4
TOTAL	2000	12	0	66	17.02	0.00%	1000.0/sec	202.50	487.20	207.4

Рисунок 7 — Результат стресс-теста в JMeter в виде таблицы

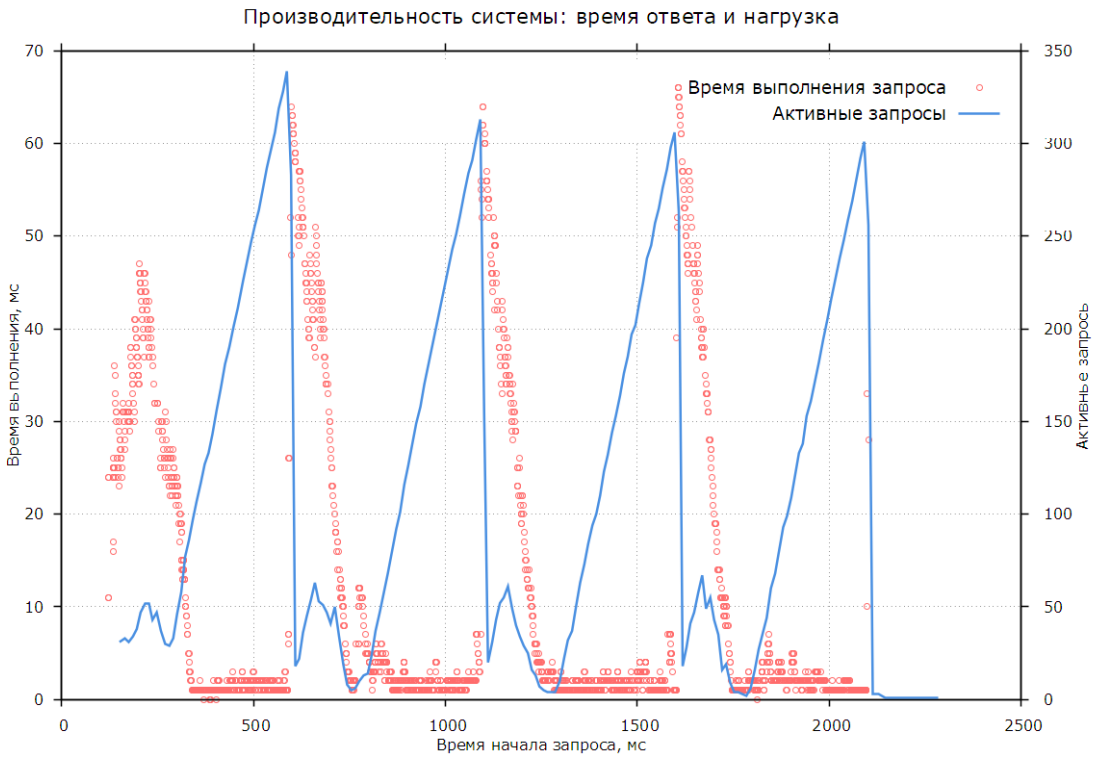

Рисунок 8 — Результат стресс-теста в JMeter в виде графика

5.4 Интеграция в систему LuNA

Подсистема интеграции была развёрнута на машине с функционирующими компонентами системы LuNA, подключена к ним и введена в эксплуатацию. Поскольку поддержаны только компоненты хранилища значений переменных и библиотеки фрагментов кода, на данный момент нельзя сказать, что подсистема интеграции полностью встроилась в систему. Для полноценного внедрения достаточно расширить поддерживаемые компоненты и реализовать для них необходимые операторы. В данный момент подсистема интеграции активно используется пользовательским интерфейсом, который также запущен на стороне подсистемы интеграции и имеет путь запроса /ui.

ЗАКЛЮЧЕНИЕ

Для системы LuNA была разработана и реализована подсистема интеграции компонентов. Реализована поддержка компонентов и логика управления запросами. Решение интегрировано в систему LuNA, было произведено его тестирование. Разработанное решение может быть применено для обеспечения коммуникации между компонентами системы LuNA, а также в распределённых системах с разнородными модулями, включая облачные платформы и IoT-решения.

На защиту выносятся следующие положения:

– Разработан формат запросов передачи данных;

– Разработана архитектура подсистемы интеграции, обеспечивающая расширение поддерживаемых компонентов и протоколов передачи данных;

– Разработаны операторы, основанные на событийном подходе и обеспечивающие возможность гибкой настройки логики системы.

Разработанная подсистема интеграции в дальнейшем может развиваться в направлении расширения списка поддерживаемых компонентов и запросов, разработки операторов и улучшения их алгоритмов, добавления новых событий в системе и улучшения архитектуры взаимодействия между оператором и подсистемой интеграции.

Выпускная квалификационная работа выполнена мной самостоятельно и с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения (концепции) из опубликованной научной литературы и других источников имеют ссылки на них. Я несу ответственность за приведенные данные и сделанные выводы.

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного цитирования является основанием для недопуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

_____ _____

ФИО студента *Подпись студента*

« ____ »_____20 __г.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. HTTP | MDN. - URL: https://developer.mozilla.org/ru/docs/Web/HTTP (дата обращения 12.01.2025).

2. JSON. - URL: https://www.json.org/json-ru.html (дата обращения 05.02.2025).

3. Extensible Markup Language (XML) 1.0 (Fifth Edition). - URL: https://www.w3.org/TR/xml (дата обращения 18.03.2025).

4. Core concepts, architecture and lifecycle | gRPC. - URL: https://grpc.io/docs/what-is-grpc/core-concepts (дата обращения 22.01.2025).

5. Protocol Buffers Documentation. - URL: https://protobuf.dev (дата обращения 09.04.2025).

6. RFC 7540 - Hypertext Transfer Protocol Version 2 (HTTP/2). - URL: https://httpwg.org/specs/rfc7540.html (дата обращения 14.02.2025).

7. RabbitMQ Documentation | RabbitMQ. - URL: https://www.rabbitmq.com/docs (дата обращения 30.03.2025).

8. AMQP is the Internet Protocol for Business Messaging | AMQP. - URL: https://www.amqp.org/about/what (дата обращения 11.01.2025).

9. Kong Gateway | Kong Docs. - URL: https://docs.konghq.com/gateway (дата обращения 27.04.2025).

10. nginx. - URL: https://nginx.org/ru/ (дата обращения 03.02.2025).

11. PostgreSQL: The world's most advanced open source database. - URL: https://www.postgresql.org (дата обращения 19.04.2025).

12. Малышкин В.Э., Перепёлкин В.А. Построение баз активных знаний для автоматического конструирования решений прикладных задач на основе системы LuNA // Параллельные вычислительные технологии – XVIII всероссийская научная конференция с международным участием, ПаВТ’2024, г. Челябинск, 2–4 апреля 2024 г. Короткие статьи и описания плакатов. Челябинск: Издательский центр ЮУрГУ, 2024. с. 57-68. DOI: 10.14529/pct2024

13. Documentation | Node.js. - URL: https://nodejs.org/en/docs (дата обращения 08.03.2025).

14. TypeScript: JavaScript With Syntax For Types. - URL: https://www.typescriptlang.org (дата обращения 25.01.2025).

15. Introduction | Fastify. - URL: https://fastify.dev/docs/latest (дата обращения 17.02.2025).

16. Node | gRPC. - URL: https://grpc.io/docs/languages/node (дата обращения 06.04.2025).

17. Socket.IO. - URL: https://socket.io (дата обращения 12.04.2025).

18. Git. - URL: https://git-scm.com (дата обращения 29.03.2025).

19. GitHub · Build and ship software on a single, collaborative platform · GitHub. - URL: https://github.com (дата обращения 10.02.2025).

20. Apache JMeter - Apache JMeter™. - URL: https://jmeter.apache.org (дата обращения 23.04.2025).

ПРИЛОЖЕНИЕ А

Программа имитации пользовательского интерфейса для теста

```
import {Logger} from "../utils/logger";
import {runKiller} from "../utils/killer";
import {serverConfigs} from "../configs/serverConfigs";
import {delay} from "../utils/delay";

export const uiLogger = new Logger(__dirname, "UI");

async function getRequest(url: string): Promise<string> {
  const form = await (await fetch(url)).formData();
  const data = form.get("data");
  if(!data || typeof data === "object") {
    await uiLogger.info("Data field is invalid");
    throw new Error("Data field is invalid");
  }
  return data;
}

async function deleteRequest(url: string): Promise<void> {
  await fetch(url, {method: "DELETE"});
  return;
}

(async function() {
  await runKiller(__dirname);
  await uiLogger.info("Running UI...");

  const serverConfig = serverConfigs.restApiServer;

  const startTime = Date.now();
  await uiLogger.info("///  1. Setting up variable values ///");
  for(let i = 0; i < 5; i++) {
    const formData  = new FormData();
    formData.set("data", new Blob([`Variable value ${i}`], {
      type: "application/octet-stream",
    }))
    const resp = await (await fetch(`http://${serverConfig.host}:${serverConfig.port}/api/v2/var-
storage/value`, {
      method: "POST",
      body: formData
    })).json();
    await uiLogger.info("New value ID: ", resp.id);
    await delay(10);
  }

  await uiLogger.info("///  2. Variable values list request ///");
  const listResp1 = await getRequest(`http://${serverConfig.host}:${serverConfig.port}/api/v2/var-
storage/list`);
  const list1: string[] = JSON.parse(listResp1);
  await uiLogger.info("First list: ", list1);

  await uiLogger.info("///  3. Variable values fetch ///");
  await Promise.all(list1.map(async (id) => {
    const resp = await getRequest(`http://${serverConfig.host}:${serverConfig.port}/api/v2/var-
storage/value/${id}`);
    await uiLogger.info(`ID: ${id}, value: ${resp}`);
```

```
  }))

  await uiLogger.info("///  4. Variable values deleting ///");
  await Promise.all(list1.map(async (id) => {
    const resp = await deleteRequest(`http://${serverConfig.host}:${serverConfig.port}/api/v2/var-storage/value/${id}`);
    await uiLogger.info(`Deleted: ${id}`);
  }))

  await uiLogger.info("///  5. Variable values list request #2 ///");
  const listResp2 = await getRequest(`http://${serverConfig.host}:${serverConfig.port}/api/v2/var-storage/list`);
  const list2: string[] = JSON.parse(listResp2);
  await uiLogger.info("Second list: ", list2);
  await uiLogger.info("/////  Execution time:", (Date.now() - startTime)/1000);
})()
```

ПРИЛОЖЕНИЕ Б

Сокращённый журнал работы сценария тестирования оператора хранилищ значений переменных:

```
[INFO] [R] [19:11:36]  Starting Core...

[INFO] [CORE] [19:11:38]  Endpoints are initiating
[INFO] [CORE] [19:11:38]  The endpoint "variableStorage1" connected to host http://0.0.0.0:6001
[INFO] [CORE] [19:11:38]  The endpoint "variableStorage2" connected to host http://0.0.0.0:6002
[INFO] [CORE] [19:11:38]  The endpoint "variableStorage3" connected to host http://0.0.0.0:6003
[INFO] [CORE] [19:11:38]  Endpoints init finished
[INFO] [CORE] [19:11:38]  Servers are starting
[INFO] [CORE] [19:11:38]  The server "operatorsServer" started on port 5053
[INFO] [CORE] [19:11:38]  The server "restApiServer" started on port 5051
[INFO] [CORE] [19:11:38]  Servers start finished

[INFO] [R] [19:11:40]  Starting Endpoints...

[INFO] [ENDPOINTS] [19:11:41]  Running endpoints...
[INFO] [ENDPOINTS] [19:11:41]  Server running at http://0.0.0.0:6001/valst
[INFO] [ENDPOINTS] [19:11:41]  Server running at http://0.0.0.0:6002/valst
[INFO] [ENDPOINTS] [19:11:41]  Server running at http://0.0.0.0:6003/valst

[INFO] [R] [19:11:43]  Starting Operators...

[INFO] [OPERATORS] [19:11:44]  Running operators...
[INFO] [OPERATORS] [19:11:44]  Connected
[INFO] [OPERATORS] [19:11:44]  [varst operator] Event: subscribe

[INFO] [R] [19:11:45]  Starting UI...

[INFO] [UI] [19:11:46]  Running UI...
[INFO] [UI] [19:11:46]  ///  1. Setting up variable values ///
[INFO] [OPERATORS] [19:11:46]  [varst operator] Event: new-request SET VAR_VALUE
[INFO] [OPERATORS] [19:11:46]  [varst operator] Event: new-request-target
[INFO] [CORE] [19:11:46]  Endpoint resolved by operator
[INFO] [CORE] [19:11:46]  Request resolving result:  { endpointId: 'fcadd6af-fcf3-4ebc-923b-
c7e00813b95b' }
[INFO] [ENDPOINTS] [19:11:46]  [variableStorage3]   POST /valst/
[INFO] [UI] [19:11:46]  New value ID:  1747224706558
...
[INFO] [UI] [19:11:46]  ///  2. Variable values list request ///
[INFO] [OPERATORS] [19:11:46]  [varst operator] Event: new-request GET VAR_VALUE_LIST
[INFO] [OPERATORS] [19:11:46]  [varst operator] Event: make-request GET VAR_VALUE_LIST
[INFO] [CORE] [19:11:46]  Endpoint fixed:  117f79f9-5c2d-4920-b13b-9f2c150f598d
[INFO] [CORE] [19:11:46]  Request resolving result:  { endpointId: '117f79f9-5c2d-4920-b13b-
9f2c150f598d' }
[INFO] [ENDPOINTS] [19:11:46]  [variableStorage1]    GET /valst/
[INFO] [OPERATORS] [19:11:46]  [varst operator] Event: make-request-response with buffer
["1747224706596","1747224706655"]
...
[INFO] [OPERATORS] [19:11:46]  [varst operator] Event: new-request-response with buffer
["1747224706596","1747224706655","1747224706630","1747224706558","1747224706671"]
[INFO] [CORE] [19:11:46]  Endpoint resolved by operator
[INFO] [CORE] [19:11:46]  Request resolving
result:  ["1747224706596","1747224706655","1747224706630","1747224706558","1747224706671"]
[INFO] [UI] [19:11:46]  First list:  [
  '1747224706596',
  '1747224706655',
  '1747224706630',
  '1747224706558',
  '1747224706671'
]
[INFO] [UI] [19:11:46]  ///  3. Variable values fetch ///
[INFO] [OPERATORS] [19:11:46]  [varst operator] Event: new-request GET VAR_VALUE
[INFO] [OPERATORS] [19:11:46]  Use cached 1747224706596: 117f79f9-5c2d-4920-b13b-9f2c150f598d
[INFO] [OPERATORS] [19:11:46]  [varst operator] Event: new-request-target
[INFO] [CORE] [19:11:46]  Endpoint resolved by operator
[INFO] [CORE] [19:11:46]  Request resolving result:  { endpointId: '117f79f9-5c2d-4920-b13b-
9f2c150f598d' }
[INFO] [ENDPOINTS] [19:11:46]  [variableStorage1]    GET /valst/1747224706596
[INFO] [UI] [19:11:46]  ID: 1747224706596, value: Variable value 1
...
[INFO] [UI] [19:11:46]  ///  4. Variable values deleting ///
[INFO] [OPERATORS] [19:11:46]  [varst operator] Event: new-request SET VAR_VALUE_DELETE
```

```
        [INFO] [OPERATORS] [19:11:46]  Deleting { id: '1747224706596' } in 117f79f9-5c2d-4920-b13b-
9f2c150f598d
        [INFO] [OPERATORS] [19:11:46]  Deleting { id: '1747224706596' } in 6093d41f-8ca6-4b3c-ab41-
d43c818a0695
        [INFO] [OPERATORS] [19:11:46]  Deleting { id: '1747224706596' } in fcadd6af-fcf3-4ebc-923b-
c7e00813b95b
        [INFO] [OPERATORS] [19:11:46]  [varst operator] Event: new-request SET VAR_VALUE_DELETE
        [INFO] [OPERATORS] [19:11:46]  [varst operator] Event: new-request SET VAR_VALUE_DELETE
        [INFO] [OPERATORS] [19:11:46]  [varst operator] Event: new-request SET VAR_VALUE_DELETE
        [INFO] [CORE] [19:11:46]  Endpoint fixed:  117f79f9-5c2d-4920-b13b-9f2c150f598d
        [INFO] [CORE] [19:11:46]  Request resolving result:  { endpointId: '117f79f9-5c2d-4920-b13b-
9f2c150f598d' }
        [INFO] [ENDPOINTS] [19:11:46]  [variableStorage1] DELETE /valst/1747224706596
        [INFO] [ENDPOINTS] [19:11:46]  [variableStorage2] DELETE /valst/1747224706596
        [INFO] [ENDPOINTS] [19:11:46]  [variableStorage3] DELETE /valst/1747224706596
        [INFO] [UI] [19:11:46]  Deleted: 1747224706596
        ...
        [INFO] [UI] [19:11:46]  ///  5. Variable values list request #2 ///
        ...
        [INFO] [OPERATORS] [19:11:46]  [varst operator] Event: make-request-response with buffer []
        [INFO] [OPERATORS] [19:11:46]  [varst operator] Event: make-request-response with buffer []
        [INFO] [OPERATORS] [19:11:46]  [varst operator] Event: make-request-response with buffer []
        [INFO] [OPERATORS] [19:11:46]  [varst operator] Event: new-request-response with buffer []
        [INFO] [CORE] [19:11:46]  Endpoint resolved by operator
        [INFO] [CORE] [19:11:46]  Request resolving result:  []
        [INFO] [UI] [19:11:46]  Second list:  []

        [INFO] [UI] [19:11:46]  /////  Execution time: 0.206
        [INFO] [R] [19:11:46]  UI exited with code 0
```

ПРИЛОЖЕНИЕ В

Подсистема интеграции компонентов системы LuNA

Руководство оператора

Листов 7

Новосибирск 2025

АННОТАЦИЯ

В данном программном документе приведено руководство оператора по применению и эксплуатации подсистемы интеграции компонентов системы LuNA.

В данном программном документе, в разделе «Назначение программы» указаны сведения о назначении программы и информация, достаточная для понимания функций программы и ее эксплуатации.

В разделе «Условия выполнения программы» указаны требования, необходимые для выполнения программы.

В разделе «Выполнение программы» указана последовательность действий оператора, обеспечивающих загрузку, запуск, выполнение и завершение программы.

Оформление программного документа «Руководство оператора» произведено по требованиям ЕСПД: 19.101-77, 19.105-78, ГОСТ 19.505-79.

СОДЕРЖАНИЕ

1 НАЗНАЧЕНИЕ ПРОГРАММЫ

1.1 Функциональное назначение программы

Разработанная программа предназначена для обращения к компонентам системы LuNA по сетевым интерфейсам.

1.2 Эксплуатационное назначение программы

Подсистема интеграции должна использоваться разработчиками компонентов системы LuNA, в том числе пользовательского интерфейс, для взаимодействия друг с другом.

1.3 Состав функций

Программа обеспечивает возможность выполнения запроса к одному из поддерживаемых компонентов системы LuNA: библиотека фрагментов кода и хранилище значений переменных. Перечень доступных запросов представлен ниже:

Библиотека фрагментов кода

- Запрос списка фрагментов кода
- Запрос информации о фрагменте кода
- Добавить фрагмент кода
- Получить файлы фрагмента кода
- Получить обработанный плагином фрагмент кода
- Получить список плагинов ФК
- Добавить плагин

Хранилище значений переменных

- Получить список значений переменных
- Получить значение переменной
- Добавить значение переменной
- Удалить значение переменной

2 УСЛОВИЯ ВЫПОЛНЕНИЯ ПРОГРАММЫ

2.1 Минимальный состав аппаратных средств

Программа предназначена для использования на серверном компьютере. Устройство, на котором запускается программа, должно иметь:

- Оперативная память объемом не менее 4 Гб;
- Жёсткий диск объёмом не менее 64 Гб;
- Клавиатура.

2.2 Минимальный состав программных средств

- Операционная система, для которой существует реализация интерпретатора языка JavaScript Node.Js;
- Интерпретатор языка JavaScript Node.Js версии не ниже 18.17.1;
- Командный интерпретатор bash.

2.3 Требование к персоналу

Конечный пользователь программы (оператор) должен обладать практическими навыками использования интерфейса командной строки для запуска программы, а также навыками выполнения сетевых запросов по протоколу HTTP для использования программы.

3 ВЫПОЛНЕНИЕ ПРОГРАММЫ

3.1 Загрузка и запуск программы

Для установки зависимостей программы необходимо выполнить команду «npm install» в рабочей директории подсистемы интеграции. Запуск программы осуществляется вызовом команды «npm run start» в той же директории.

3.2 Выполнение программы

После запуска программа выполнит инициализацию, проведёт запуск необходимых элементов и будет готова к использованию. После этого, если в системе также запущены требуемые компоненты, оператор может выполнить следующие HTTP запросы, отправленные по адресу подсистемы интеграции http://localhost:5051:

Таблица 1 – HTTP запросы подсистемы интеграции

Описание	**Запрос**	**Тело запроса**	**Ответ**
Запрос списка фрагментов кода	GET /api/v2/code-f-storage /list		200 **Multipart: data** octet-stream: json список
Запрос информации о фрагменте кода	GET /api/v2/code-f-storage /info/{id}		200 **Multipart: data** octet-stream json описание
Добавить фрагмент кода	POST /api/v2/code-f-storage /fragment/{id}	**Multipart: data** octet-stream архив tar	204 *No content*
Получить файлы фрагмента кода	GET /api/v2/code-f-storage /fragment/{id}		200 **Multipart: data** octet-stream архив tar
Получить обработанный плагином фрагмент кода	GET /api/v2/code-f-storage /procedure/{id}?type="плагин"		200 **Multipart: data** octet-stream json с фрагментом кода
Получить список плагинов ФК	GET /api/v2/code-f-storage /plugins-list		200 **Multipart: data** octet-stream json список
Получить плагин	GET /api/v2/code-f-storage /plugin/{id}		200 **Multipart: data** octet-stream архив tar
Добавить плагин	POST /api/v2/code-f-storage /plugin/{id}	**Multipart: data** octet-stream архив tar	204 *No content*

Описание	Запрос	Тело запроса	Ответ
Получить список значений переменных	GET /api/v2/var-storage /list		200 **Multipart: data** octet-stream json список
Получить значение переменной	GET /api/v2/var-storage /value/{id}		200 **Multipart: data** octet-stream массив байт
Добавить значение переменной	POST /api/v2/var-storage /value	**Multipart: data** octet-stream массив байт	201 application/json {id: string}
Удалить значение переменной	DELETE /api/v2/var-storage /value/{id}		204 *No content*

3.3 Завершение работы программы

Программа работает в режиме сервера, то есть не имеет благополучного сценария завершения работы. Программу можно прервать в любой момент, используя сигнал SIGINT. Для этого можно использовать сочетание клавиш Ctrl+C.