

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
GRADUATION THESIS**

**Инструменты для диагностики распределённых систем с соблюдением требований
PCI-DSS**

Обучающийся / Student Нуруллаев Даниил

Факультет/институт/кластер/ Faculty/Institute/Cluster факультет программной инженерии и компьютерной техники

Группа/Group P34121

Направление подготовки/ Subject area 09.03.04 Программная инженерия

Образовательная программа / Educational program Системное и прикладное программное обеспечение 2020

Язык реализации ОП / Language of the educational program Русский

Квалификация/ Degree level Бакалавр

Руководитель ВКР/ Thesis supervisor Гаврилов Антон Валерьевич, Университет ИТМО, факультет программной инженерии и компьютерной техники, старший преподаватель (квалификационная категория "старший преподаватель")

Обучающийся/Student

Документ подписан	
Нуруллаев Даниил	
18.05.2024	

(эл. подпись/ signature)

Нуруллаев
Даниил

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Гаврилов Антон Валерьевич	
14.05.2024	

(эл. подпись/ signature)

Гаврилов Антон
Валерьевич

(Фамилия И.О./ name
and surname)

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ /
OBJECTIVES FOR A GRADUATION THESIS

Обучающийся / Student Нуруллаев Даниил

Факультет/институт/кластер/ Faculty/Institute/Cluster факультет программной инженерии и компьютерной техники

Группа/Group P34121

Направление подготовки/ Subject area 09.03.04 Программная инженерия

Образовательная программа / Educational program Системное и прикладное программное обеспечение 2020

Язык реализации ОП / Language of the educational program Русский

Квалификация/ Degree level Бакалавр

Тема ВКР/ Thesis topic Инструменты для диагностики распределённых систем с соблюдением требований PCI-DSS

Руководитель ВКР/ Thesis supervisor Гаврилов Антон Валерьевич, Университет ИТМО, факультет программной инженерии и компьютерной техники, старший преподаватель (квалификационная категория "старший преподаватель")

Характеристика темы ВКР / Description of thesis subject (topic)

Тема в области фундаментальных исследований / Subject of fundamental research: нет / not

Тема в области прикладных исследований / Subject of applied research: да / yes

Основные вопросы, подлежащие разработке / Key issues to be analyzed

1) Техническое задание:

Разработка инструмента для сбора профилей и дамповых данных в условиях ограничений PCI-DSS.

2) Исходные данные к работе:

1. Документация к языку программирования Java. – URL: <https://docs.oracle.com/en/java> (дата обращения 11.02.2024)

2. Документация к языку программирования Go. – URL: <https://go.dev/doc> (дата обращения 11.02.2024)

3. Документация к реляционной базе данных PostgreSQL. – URL: <https://www.postgresql.org/docs> (дата обращения 11.02.2024)

4. Документация к брокеру сообщений Apache Kafka. – URL: <https://kafka.apache.org/documentation> (дата обращения 11.02.2024)

5. Документация к фреймворку Kora. - URL: <https://kora-projects.github.io/kora-docs> (дата обращения 11.02.2024)

6. Требования к выпускным квалификационным работам. – URL:

<https://student.itmo.ru/files/1314> (дата обращения 11.02.2024)

7. Положение о выпускных квалификационных работах. – URL: <https://edu.itmo.ru/files/344> (дата обращения 11.02.2024)

3)Цель работы:

Предоставить пользователям инструмент для сбора профилей и дампов данных с целью дальнейшего улучшения их продуктов, выявления утечек памяти и анализа производительности.

4)Перечень вопросов, подлежащих разработке:

1. Анализ требований;
2. Разработка архитектуры системы;
3. Разработка системы;
4. Тестирование.

Форма представления материалов ВКР / Format(s) of thesis materials:

Презентация

Дата выдачи задания / Assignment issued on: 17.10.2023

Срок представления готовой ВКР / Deadline for final edition of the thesis 25.05.2024

СОГЛАСОВАНО / AGREED:

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Гаврилов Антон Валерьевич	
30.04.2024	

(эл. подпись)

Гаврилов Антон
Валерьевич

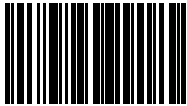
Задание принял к
исполнению/ Objectives
assumed BY

Документ подписан	
Нуруллаев Даниил	
11.05.2024	

(эл. подпись)

Нуруллаев
Даниил

Руководитель ОП/ Head
of educational program

Документ подписан	
Дергачев Андрей Михайлович	
22.05.2024	

(эл. подпись)

Дергачев
Андрей
Михайлович

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
SUMMARY OF A GRADUATION THESIS**

Обучающийся / Student Нуруллаев Даниил

Факультет/институт/кластер/ Faculty/Institute/Cluster факультет программной инженерии и компьютерной техники

Группа/Group P34121

Направление подготовки/ Subject area 09.03.04 Программная инженерия

Образовательная программа / Educational program Системное и прикладное программное обеспечение 2020

Язык реализации ОП / Language of the educational program Русский

Квалификация/ Degree level Бакалавр

Тема ВКР/ Thesis topic Инструменты для диагностики распределённых систем с соблюдением требований PCI-DSS

Руководитель ВКР/ Thesis supervisor Гаврилов Антон Валерьевич, Университет ИТМО, факультет программной инженерии и компьютерной техники, старший преподаватель (квалификационная категория "старший преподаватель")

**ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
DESCRIPTION OF THE GRADUATION THESIS**

Цель исследования / Research goal

Предоставить пользователям инструмент для сбора профилей и дампов данных с целью дальнейшего улучшения их продуктов, выявления утечек памяти и анализа производительности.

Задачи, решаемые в ВКР / Research tasks

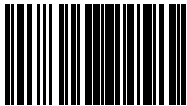
1. Анализ требований для системы диагностики. 2. Проектирование архитектуры системы для сбора артефактов приложений. 3. Реализация системы на основе спроектированной архитектуры. 4. Тестирование решения.

Краткая характеристика полученных результатов / Short summary of results/findings

В результате анализа требований была разработана архитектура системы. Для управления артефактами приложения успешно реализованы три микросервиса, каждый из которых выполняет специализированные функции в рамках общей системы. Проведено интеграционное тестирование, которое позволило проверить взаимодействие между микросервисами, и ручное тестирование, направленное на обеспечение качества конечного продукта.

Обучающийся/Student

Документ подписан	
----------------------	--

	
Нуруллаев Даниил	
18.05.2024	

(эл. подпись/ signature)

Нуруллаев
Даниил

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Гаврилов Антон Валерьевич	
14.05.2024	

(эл. подпись/ signature)

Гаврилов Антон
Валерьевич

(Фамилия И.О./ name
and surname)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	3
1 Обзор аналогов и анализ требований.....	4
1.1 Обзор аналогов	4
1.2 Анализ требований.....	4
2 Разработка архитектуры системы сбора артефактов приложений	5
2.1 Архитектура системы	5
2.2 Выбор стека технологий.....	7
2.3 Схема базы данных	11
2.4 Выбор способа общения.....	12
3 Разработка системы сбора артефактов приложений	18
3.1 Подготовка окружения для разработки	18
3.2 Разработка агента	21
3.3 Разработка доставщика команд	25
3.4 Разработка сборщика артефактов.....	27
3.5 Разработка менеджера	29
4 Тестирование	31
4.1 Интеграционное тестирование	31
4.2 Ручное тестирование.....	32
ЗАКЛЮЧЕНИЕ	34
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	35

ВВЕДЕНИЕ

В современном мире информационных технологий, где данные становятся всё более ценными, обеспечение безопасности персональных данных клиентов является ключевым приоритетом для всех организаций. Особое значение приобретает соблюдение стандартов безопасности в разработке и эксплуатации информационных систем. Один из таких стандартов, Payment Card Industry Data Security Standard (PCI-DSS), направлен на защиту данных о платежных картах и информации, связанной с платёжными операциями.

Существующий стандарт вносит ограничения на эксплуатацию Kubernetes, что делает невозможным сбор дампов памяти и профилей выполнения приложений, функционирующих внутри Kubernetes. Для обеспечения соблюдения требований PCI-DSS и одновременной возможности анализа и оптимизации работы приложений, необходимо разработать специализированное решение. Это решение должно позволять собирать необходимые артефакты, не нарушая установленных стандартом рамок безопасности.

1 Обзор аналогов и анализ требований

1.1 Обзор аналогов

В ходе исследования данной области не было обнаружено открытых аналогов предлагаемого решения. Отсутствие таких аналогов можно объяснить тем, что разработка приложения тесно связана с политикой безопасности компаний. Создать универсальное решение, которое одновременно соответствовало бы требованиям безопасности всех потенциальных пользователей, представляется крайне сложной задачей.

1.2 Анализ требований

Перед тем как приступить к разработке архитектуры системы, важно выделить все важные требования к системе.

1. **Инструментарий профилирования:** Система должна предоставлять инструменты для сбора профилей, дампов памяти и дампов потока JVM-приложений. Это позволит анализировать производительность приложений и выявлять узкие места в работе приложений.
2. **Обфускация данных:** Важной функцией системы является возможность обфускации данных в дампах памяти, что необходимо для соответствия стандартам безопасности данных PCI-DSS. Это обеспечит защиту конфиденциальной информации в процессе анализа дампов.
3. **Хранение артефактов:** Система должна обладать функционалом для сохранения артефактов приложений.
4. **Система авторизации:** Для обеспечения безопасности, система должна включать механизм авторизации пользователей. Это позволит ограничить и управлять правами пользователей на выполнение операций в зависимости от их роли.

2 Разработка архитектуры системы сбора артефактов приложений

2.1 Архитектура системы

Первостепенно нужно продумать, как будет работать решение. Суть проблемы заключается в том, что наши возможности в Kubernetes ограничены, что и приводит к возникновению проблемы. Поскольку каждое приложение в Kubernetes работает в отдельном контейнере, решение должно иметь возможность попасть в контейнер вместе с приложением, чтобы инициировать сбор артефактов. В контексте JVM-приложений лучшим решением будет написание Java-агента. Java Agent — это компонент экосистемы Java-приложений, который можно описать как дочерний процесс, работающий вместе с основным приложением. Благодаря такому агенту можно будет решить проблему доступа к контейнеру, так как агент будет находиться в том же контейнере как дочерний процесс.

Определившись с тем, как реализовать инициацию сбора артефактов, необходимо выбрать инструмент, который будет использоваться для сбора профилей и дампов. Выбор был сделан в пользу jcmd. Самый важный плюс jcmd [1] заключается в том, что он объединяет в себе весь необходимый функционал для сбора профилей, дампа потоков и дампов памяти, в отличие от других технологий, таких как jstack, jinfo, jmap [2, 3, 4] и т.д. Также важно отметить, что jcmd использует те же механизмы управления, что и другие средства JVM, и его использование оказывает меньшее воздействие на производительность приложения.

Теперь важно понять, как конечные пользователи будут взаимодействовать с этим агентом. Для этого было принято решение разработать отдельное приложение-посредник между агентом и пользователями. В качестве архитектуры для приложения был сделан выбор в пользу микросервисной архитектуры. Данный тип архитектуры легче всего поддерживать, разрабатывать и масштабировать. Она имеет более высокую

отказоустойчивость по сравнению с монолитной архитектурой, поскольку сбой одного микросервиса не влечет за собой сбой всей системы. Также время старта приложения у микросервисов всегда меньше за счёт их размера. Приложение будет состоять из трех микросервисов:

1. Сборщик артефактов: Микросервис, отвечающий за работу с артефактами, полученными от агентов.
2. Доставщик команд: Микросервис, отвечающий за работу с командами, которые приходят от пользователей для запуска сбора артефактов.
3. Менеджер: Микросервис, отвечающий за работу с квотами для хранения полученных артефактов.

В качестве инструмента авторизации было решено использовать Open Policy Agent (OPA) [5], а в качестве аутентификации — внутренний ресурс. Выбор был сделан в пользу OPA из-за его масштабируемости и удобства. Он позволяет описывать все политики внутри одного приложения, что упрощает внедрение, поскольку при необходимости нужно просто добавить новую политику в конфигурацию OPA.

На основе всего описанного ранее была составлена архитектура всей системы, которая представлена на рисунке 1.

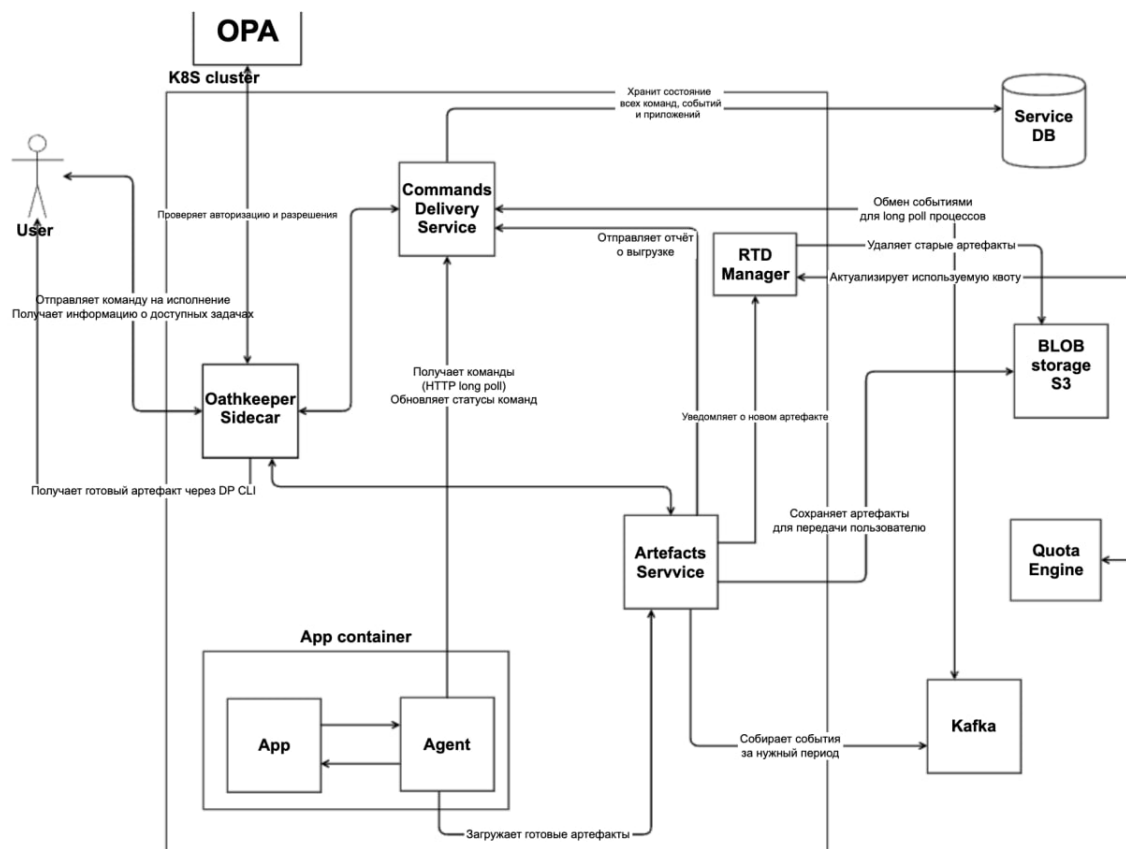


Рисунок 1 — Архитектура системы

2.2 Выбор стека технологий

После проектирования архитектуры системы необходимо выбрать технологии, которые будут использоваться для её реализации.

Для создания агента используется Java [6], поскольку Java-агент может функционировать исключительно в среде JVM. Тем не менее, было решено учесть будущие возможности масштабирования. Рано или поздно потребуется расширить функционал системы и собирать артефакты с приложений, написанных на различных языках программирования. Поскольку технология Java Agent поддерживается только в JVM, для других языков она не подходит. Исходя из предположения, что агент будет работать как дочерний процесс, в других языках программирования его можно будет запускать как обычный дочерний процесс. В таком случае наиболее подходящим языком для реализации функционала запуска и сбора является язык, компилирующийся в

бинарный файл. Среди современных языков программирования наилучшим выбором представляются C++, Rust и Go. Наиболее подходящим оказался Go [7], поскольку он имеет ряд преимуществ. Go включает в себя сборщик мусора, что значительно упрощает разработку. Синтаксис Go более интуитивен по сравнению с C++ и Rust. Благодаря этим преимуществам, можно сделать вывод, что Go будет наилучшим решением для быстрой и надёжной разработки. Реализовав весь необходимый функционал на Go, агент будет запускать этот бинарный файл, который в дальнейшем можно будет использовать и для других языков.

Для реализации микросервисов был выбран язык Java. Создание серверной части на Java обладает множеством преимуществ. Java обеспечивает высокую производительность и надёжность, что делает её подходящей для масштабируемых и высоконагруженных систем. Её независимость от платформ позволяет запускать программы на любой платформе с JVM. Java обладает обширной экосистемой с множеством библиотек и фреймворков, что упрощает и ускоряет разработку. Безопасность встроена в язык, а большое сообщество разработчиков и удобные инструменты разработки (например, IntelliJ IDEA) делают процесс создания приложений более эффективным.

Создание современных веб-приложений без фреймворка крайне затруднительно, поэтому был выбран фреймворк для реализации. Выбор пал на фреймворк Kora [8]. Kora — это фреймворк для разработки приложений на Java и Kotlin, который фокусируется на производительности, эффективности и прозрачности. Kora предоставляет высокоуровневые декларативные инструменты и абстракции, которые на этапе компиляции преобразуются в производительный и понятный код. Преимущества Kora перед другими фреймворками:

1. Производительность: Kora генерирует высокопроизводительный код на этапе компиляции, исключает использование Reflection API и динамических прокси, реализует тонкие абстракции и аспекты, что

обеспечивает высокую производительность приложений, низкое время отклика и возможность обработки большого количества запросов в секунду.

2. Эффективность: Благодаря вышеперечисленным факторам и тому, что контейнер зависимостей создается на этапе компиляции и инициализируется максимально параллельно, достигается низкое время старта. Это позволяет эффективно использовать практики горизонтального масштабирования и максимально утилизировать ресурсы не только в рамках приложения, но и всего кластера.
3. Прозрачность: Kora генерирует читаемый исходный код на этапе компиляции, что вместе с тонкими абстракциями и аспектами обеспечивает высокую читаемость кода и понимание основных механизмов работы фреймворка со стороны разработчика.

Сравнение производительности Kora над другими фреймворками представлено на рисунках 2 и 3.

Для хранения артефактов приложений было решено использовать S3, так как это единственное решение, подходящее для удобного сохранения и получения файлов. В качестве хранилища данных, необходимых для работы приложений, лучшим выбором оказалась PostgreSQL [9] за её надёжность, безопасность и строгое соответствие стандартам SQL.

Параметры тестирования				
	Spring Boot + JPA	Spring Boot + JDBC Template	Spring + JDBC Template	Kora + JDBC
Bootstrap память (минимальная)	192 mb	160 mb	128 mb	64 mb
Bootstrap время (минимальное)	16s	10s	3.5s	3.2s
Потребление CPU при 500 RPS	63%	48%	42%	40%
RPS (максимальный)	710	1034	1367	1657
Среднее время ответа	2.39 ms	2.21 ms	1.79 ms	1.7 ms

Условия тестирования	
RAM	384 Mb
CPU	0.5 CPU (пол ядра)
Java	17 OpenJDK
Сценарий	1 POST + 2 GET + 1 DELETE

Результаты тестирования	
Bootstrap память	Kora потребляет на 300% меньше
Bootstrap время	Kora стартует на 400% быстрее
RPS	Kora обрабатывает в 2.3 раза больше RPS
Среднее время ответа	Kora имеет на 40% лучший Latency

Рисунок 2 — Сравнение производительности Kora и Spring

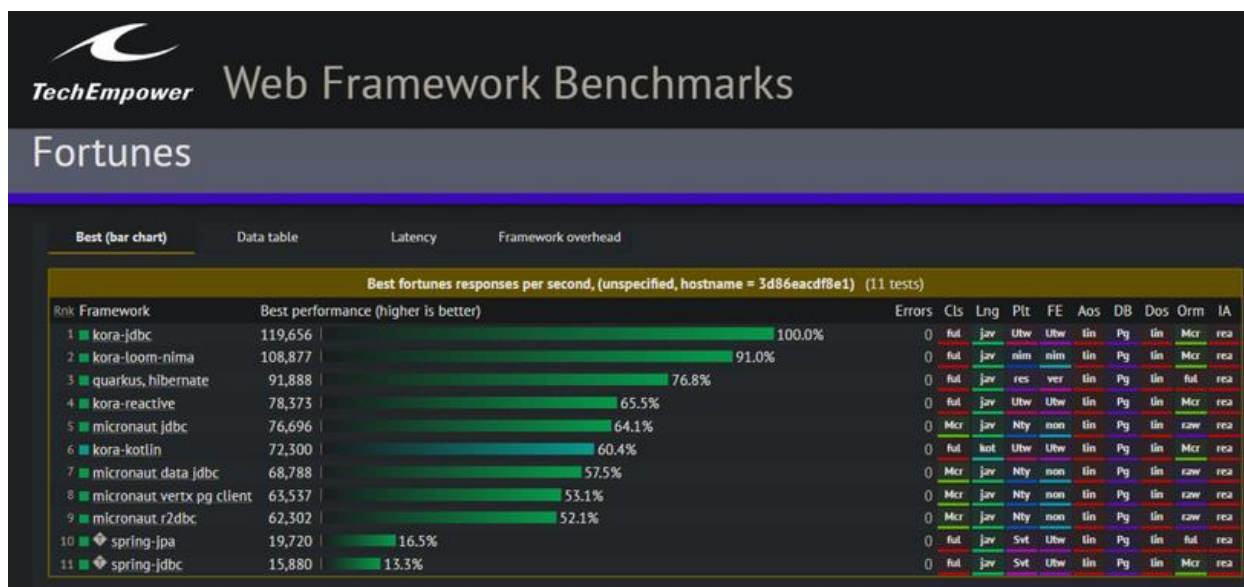


Рисунок 3 — Сравнение производительности Kora и других фреймворков

2.3 Схема базы данных

Определившись с выбором технологий, было решено составить схему базы данных, рисунок 4.

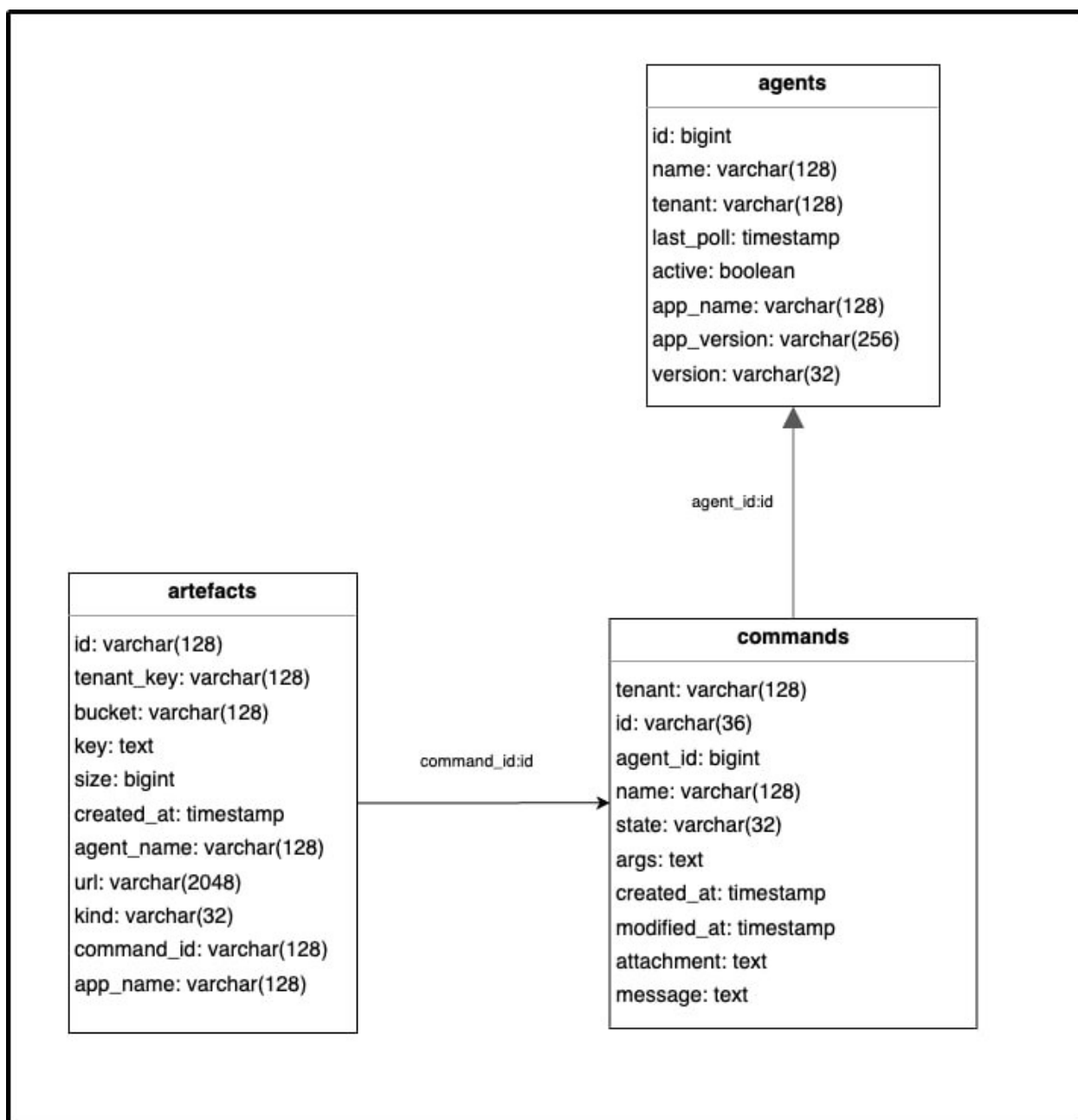


Рисунок 4 — Схема базы данных

На схеме представлены три таблицы:

1. Таблица агентов: В этой таблице хранится вся информация об агентах и приложениях, с которыми внедрен агент.

2. Таблица команд: В данной таблице содержится информация о командах, которые отправляются на агента, и их состоянии.
3. Таблица артефактов: В этой таблице хранится информация обо всех артефактах, которые приходят от агентов, а также URL для их получения из S3.

2.4 Выбор способа общения

Система состоит из трёх микросервисов и агента. Поскольку все компоненты изолированы друг от друга, необходимо тщательно продумать, как пользователь будет взаимодействовать с системой и как сервисы будут общаться между собой. На сегодняшний день наиболее популярными подходами для взаимодействия являются gRPC, RESTful и брокеры сообщений. Проведя анализ этих технологий, можно выделить их плюсы и минусы:

gRPC:

- Плюсы:
 - Высокая производительность: Благодаря бинарному формату данных и использованию HTTP/2.
 - Синхронное и асинхронное взаимодействие: Поддерживает оба типа, что делает его гибким для различных задач.
 - Многоязычная поддержка: Доступен для множества языков программирования.
- Минусы:
 - Сложность настройки: Требуется генерация кода и использование .proto файлов.
 - Меньшая читаемость: Бинарный формат сложнее читать и отлаживать вручную по сравнению с текстовыми форматами.

RESTful:

- Плюсы:
 - Простота и широкая поддержка: Легко настроить и использовать, хорошо известен и поддерживается многими языками и фреймворками.
 - Человеческая читаемость: Данные передаются в текстовом формате (JSON, XML), что упрощает отладку и понимание.
 - Стандартность: Хорошо подходит для публичных API и веб-сервисов.
- Минусы:
 - Средняя производительность: Из-за использования текстовых форматов.

Броке сообщений:

- Плюсы:
 - Асинхронное взаимодействие: Идеально для задач, где требуется асинхронная обработка сообщений.
 - Высокая производительность: Хорошо подходит для обработки большого количества событий и сообщений.
 - Гибкость: Можно использовать различные форматы данных и протоколы.
- Минусы:
 - Сложность настройки: Требуется дополнительной настройки и управления брокером сообщений.
 - Зависимость от брокера: Производительность и надежность зависят от выбранного брокера сообщений.

Учитывая все плюсы и минусы этих подходов, было решено использовать RESTful для взаимодействия пользователей с сервисами и брокеры сообщений для общения между самими сервисами.

Выбор RESTful для взаимодействия пользователей с сервисами обусловлен рядом ключевых факторов. RESTful API предоставляет удобный и широко поддерживаемый способ реализации взаимодействия между клиентом и сервером. В данном контексте это оказывается более предпочтительным по сравнению с gRPC по нескольким причинам:

1. Удобство реализации: RESTful API использует стандартные HTTP методы (GET, POST, PUT, DELETE), что упрощает его внедрение и поддержку. Большинство современных веб-фреймворков и языков программирования имеют встроенную поддержку REST, что позволяет быстро начать разработку и легко интегрировать API с клиентом.
2. Отсутствие необходимости в асинхронном общении: В данном кейсе асинхронное общение не является критически важным. RESTful API прекрасно подходит для синхронного взаимодействия, где клиент отправляет запрос и ожидает ответ от сервера. Это упрощает логику работы клиентских приложений и делает их более предсказуемыми.
3. Низкие требования к производительности: Производительность не является основным приоритетом для данного проекта. RESTful API на базе HTTP/1.1 или HTTP/2 обеспечивает достаточную производительность для большинства приложений, особенно если не требуется обработка большого объема данных в реальном времени. HTTP/2 дополнительно улучшает производительность за счет таких возможностей, как мультиплексирование запросов и сжатие заголовков.

Для общения между сервисами был выбран брокер сообщений. Это решение обусловлено несколькими ключевыми факторами, особенно учитывая, что работа идет с дампами и профилями приложений, что подразумевает передачу больших файлов. Вот почему использование брокера сообщений оказалось оптимальным:

1. Асинхронное взаимодействие:

- Обработка больших данных: Отправка больших файлов занимает значительное время, и асинхронное взаимодействие здесь более эффективно. Брокеры сообщений позволяют отправителю и получателю обрабатывать сообщения в своем темпе, не блокируя основные процессы.
- Повышенная устойчивость: Если отправитель или получатель временно недоступны, брокеры сообщений сохраняют файлы в очереди до тех пор, пока получатель не будет готов их обработать.

2. Надежность и гарантии доставки:

- Подтверждение получения: Брокеры сообщений поддерживают механизмы подтверждения доставки, гарантируя, что файлы будут доставлены даже в случае временных сбоев сети или приложений.
- Повторные попытки доставки: В случае сбоя брокеры сообщений автоматически повторяют попытку доставки, повышая надежность передачи данных.

Остаётся только определиться с выбором брокера сообщений. Среди самых популярных решений — ActiveMQ, RabbitMQ и Kafka [10].

ActiveMQ:

- Плюсы:
 - Простота настройки и использования: Легко настраивается и используется, особенно для новичков.
 - Поддержка различных протоколов: Поддерживает множество протоколов, таких как MQTT, AMQP, STOMP, OpenWire и другие.
 - Сообщения с гарантией доставки: Поддерживает надежную доставку сообщений с подтверждением и сохранением сообщений.
- Минусы:

- Производительность: Не подходит для обработки огромных объемов сообщений в реальном времени, уступает Kafka в производительности.
- Масштабируемость: Масштабируется хуже по сравнению с Kafka и RabbitMQ.
- Проблемы с консистентностью: В некоторых сценариях могут возникать проблемы с консистентностью данных.

RabbitMQ:

- Плюсы:
 - Высокая производительность: Отличная производительность и низкая задержка при обработке сообщений.
 - Гибкость маршрутизации: Поддержка сложных схем маршрутизации и обменов.
 - Поддержка различных протоколов: Поддерживает AMQP, MQTT, STOMP и другие.
 - Надежность: Поддерживает подтверждение доставки и сохранение сообщений.
- Минусы:
 - Сложность настройки: Более сложен в настройке по сравнению с ActiveMQ.
 - Масштабируемость: Масштабируемость лучше, чем у ActiveMQ, но все же уступает Kafka.
 - Управление памятью: Требуется тщательной настройки параметров управления памятью для работы с большими объемами данных.

Kafka:

- Плюсы:

- Производительность: Очень высокая производительность, подходит для обработки больших объемов данных в реальном времени.
- Масштабируемость: Легко масштабируется, поддерживает горизонтальное масштабирование.
- Устойчивость к сбоям: Высокая устойчивость к сбоям и поддержка репликации данных.
- Поддержка событийной архитектуры: Отлично подходит для реализации событийно-ориентированных систем и потоковой обработки данных.
- Минусы:
 - Сложность настройки и управления: Требуется более сложная настройка и управление по сравнению с ActiveMQ и RabbitMQ.
 - Ограниченная поддержка протоколов: Основной протокол — собственный, что может усложнять интеграцию с некоторыми системами.
 - Консистентность: Возможны задержки в консистентности данных, особенно при больших нагрузках и репликации.

Выбор был сделан в пользу Kafka, учитывая её многочисленные преимущества. Kafka лучше всего подходит для передачи большого объема данных и обеспечивает самую высокую надёжность в доставке сообщений, что идеально соответствует задачам, связанным с отправкой файлов.

3 Разработка системы сбора артефактов приложений

3.1 Подготовка окружения для разработки

Прежде чем начать разработку, важно позаботиться о создании тестового окружения. Для разработки понадобятся такие сервисы, как PostgreSQL, Kafka и S3. Наиболее удобным способом развертывания всех необходимых сервисов является использование контейнеров. В качестве инструмента контейнеризации был выбран Docker Compose. Docker Compose — это инструмент, который упрощает управление всем стеком приложений, которые помещаются в контейнеры.

Для того чтобы поднять все нужные сервисы, необходимо создать файл в формате YAML с описанием этих сервисов. Поднятие всех сервисов приведено в листинге 1.

```
version: '3.8'

networks:
  test-net:
    driver: bridge

services:
  postgresql:
    image: "postgres:11.11"
    restart: "always"
    ports:
      - "5432:5432"
    networks:
      - test-net
    environment:
      POSTGRES_USER: "rtd"
      POSTGRES_PASSWORD: " rtd "
      POSTGRES_DB: " rtd "
    volumes:
      - postgres-data:/var/lib/postgresql/data

  zookeeper:
    image: "docker-proxy.artifactory.tcsbank.ru/zookeeper:3.4.9"
    hostname: zookeeper
    ports:
      - "2181:2181"
    networks:
      - test-net
    environment:
```

```

ZOO_MY_ID: 1
ZOO_PORT: 2181
ZOO_SERVERS: server.1=zoo1:2888:3888

kafka:
  image: "confluentinc/cp-kafka:5.5.1"
  hostname: kafka
  networks:
    - test-net
  ports:
    - "9092:9092"
    - "9093:9093"
  environment:
    KAFKA_ADVERTISED_LISTENERS: BROKER://kafka:9092,
EXTERNAL://localhost:9093
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
BROKER:PLAINTEXT,EXTERNAL:PLAINTEXT
    KAFKA_INTER_BROKER_LISTENER_NAME: BROKER
    KAFKA_ZOOKEEPER_CONNECT: "zookeeper:2181"
    KAFKA_BROKER_ID: 1
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
  depends_on:
    - zookeeper

s3:
  image: "localstack/localstack:0.12.15"
  hostname: s3
  networks:
    - test-net
  ports:
    - "4566:4566"
  environment:
    - USE_SSL=false
    - SERVICES=s3
    - DEBUG=1
  volumes:
    - localstack-data:/var/lib/localstack

volumes:
  postgres-data:
  localstack-data:

```

Листинг 1 — Поднятие окружения

Подняв окружение, осталось только создать таблицы в базе данных. Для создания и последующего изменения таблиц было решено использовать миграции. В качестве инструмента для миграций был выбран Flyway. Для каждой таблицы были созданы три отдельные версии миграций, которые представлены в листингах 2, 3 и 4.

```

create table agents
(
    id bigserial not null,
    name varchar(128) not null,
    tenant varchar(128) not null,
    last_poll timestamp not null default current_timestamp,
    active boolean not null default true,
    app_name varchar(128) not null,
    app_version varchar(256) not null,
    version varchar(32) not null
);

create unique index agents_pk_idx on agents (id);
create index agents_name_idx on agents using hash (name);
create index agents_tenant_idx on agents using hash (tenant);
create index agents_last_poll_idx on agents using btree (last_poll);

```

Листинг 2 — Создание таблицы агентов

```

create table commands
(
    tenant varchar(128) not null,
    id varchar(36) not null,
    agent_id bigint not null,
    name varchar(128) not null,
    state varchar(32) not null,
    args text not null,
    created_at timestamp default current_timestamp,
    modified_at timestamp default null,
    attachment text default null,
    message text default null
);

create unique index commands_pk_idx on agents (id);

```

Листинг 3 — Создание таблицы команд

```

create table artefacts
(
    id varchar(128) not null,
    tenant_key varchar(128) not null,
    bucket varchar(128) not null,
    key text not null,
    size integer not null,
    created_at timestamp not null,
    agent_name varchar(128),
    url varchar(2048) not null,
    kind varchar(32) not null,
    command_id varchar(128) not null,
    app_name varchar(128) not null,
);

create unique index artefacts_pk_idx on agents (id);

```

Листинг 4 — Создание таблицы артефактов

3.2 Разработка агента

Начать разработку агента было решено с реализации обфускатора дампа, который представлен в листинге 5.

```
func obfuscateDump(buf []byte, idSize uint32) {
    for i := uint32(0); i < uint32(len(buf)); i++ {
        tag := buf[i]
        i = i + 1
        switch tag {
            case HPROF_GC_ROOT_UNKNOWN:
                i = i + idSize
            case HPROF_GC_ROOT_THREAD_OBJ:
                i = i + idSize + 8
            case HPROF_GC_ROOT_JNI_GLOBAL:
                i = i + idSize + idSize
            case HPROF_GC_ROOT_JNI_LOCAL:
                i = i + idSize + 8
            .....
            case HPROF_GC_ROOT_JAVA_FRAME:
                i = i + idSize + 8
            case HPROF_GC_OBJ_ARRAY_DUMP:
                i = i + idSize + 4
                number := binary.BigEndian.Uint32(buf[i : i+4])
                i = i + 4 + idSize + number*idSize
            case HPROF_GC_PRIM_ARRAY_DUMP:
                i = i + idSize + 4
                elements := binary.BigEndian.Uint32(buf[i : i+4])
                i = i + 4
                t := buf[i]
                i = i + 1
                typeLen := sizeByType(t, idSize)
                if t == HPROF_CHAR {
                    memset(buf, 0, i, i+typeLen*elements)
                }
                if t == HPROF_BYTE {
                    memset(buf, 0, i, i+typeLen*elements)
                }
                i = i + typeLen*elements
            i = i - 1
        }
    }
}
```

Листинг 5 — Обфускация данных

Функция `obfuscate Dump` обрабатывает буфер байтов с данными формата `HPROF` и обфусцирует содержимое массивов типов `char` и `byte`. Она перемещается по буферу, перебирая различные теги (например,

HPROF_GC_ROOT_UNKNOWN, HPROF_GC_CLASS_DUMP), пропуская соответствующие данные и обнуляя содержимое примитивных массивов. Обфускация применяется именно к этим типам данных по следующим причинам:

1. Массивы `char` часто содержат строки, которые могут включать секретную информацию (например, счета клиентов, пароли, пути к файлам).
2. Массивы `byte` также могут содержать секретную информацию, такую как двоичные данные.

Завершив написание обфускации, началась реализация функций для сбора артефактов. Рассмотрим интерфейс с функциями для реализации сбора в листинге 6.

```
type JcmdApi interface {  
    ThreadPrint(locks *bool) (string, error)  
    GcHeapDump(all *bool) (io.ReadSeekCloser, error)  
    JfrStart(name string, filename string, settings string, duration string, maxAge string, maxSize string) error  
    JfrStop(name string, filename string) error  
    JfrDump(name string, filename string, maxage string, maxsize string) error  
}
```

Листинг 6 — Интерфейс запуска команд

1. `ThreadPrint`: Функция предназначена для сбора информации обо всех потоках с их трассировкой стека. Для запуска используется параметр `"Thread.print"`.
2. `GcHeapDump`: Функция предназначена для создания дампа кучи в формате HPROF. Для запуска используется параметр `"GC.heap_dump"`.
3. `JfrStart`: Функция предназначена для запуска записи Flight Recording. Позволяет начать запись с различными параметрами настройки. Для запуска используется параметр `"JFR.start"`.

4. JfrStop: Функция предназначена для остановки записи Flight Recording.

Позволяет завершить запись, при необходимости указав имя записи и файл для сохранения. Для запуска используется параметр "JFR.stop"

5. JfrStop: Функция предназначена для сохранения данных текущей записи Flight Recording в файл. Позволяет выбрать диапазон времени и другие параметры для дампа. Для запуска используется параметр "JFR.dump"

Для того чтобы понять, как происходит запуск jcmd, рассмотрим пример реализации функции ThreadPrint, представленный в листинге 7.

```
type localJcmdApi struct {
    jcmdBinary string
    appPid      string
    logger      logging.Logger
}

func (api *localJcmdApi) ThreadPrint(locks *bool) (string, error) {
    args := []string{api.appPid, "Thread.print"}
    if locks != nil {
        args = append(args, fmt.Sprintf("-l=%t", *locks))
    }
    cmd := exec.Command(api.jcmdBinary, args...)
    stdout, err := cmd.CombinedOutput()
    if err != nil {
        return "", errors.New(err.Error() + "\n" + string(stdout))
    }
    return string(stdout), nil
}
```

Листинг 7 — Реализация функции ThreadPrint

Для того чтобы запустить любую команду с помощью jcmd, необходим PID приложения и соответствующий параметр запуска, в данном случае "Thread.print". С использованием возможностей языка Go запускаем jcmd в командной строке и передаем ему все необходимые аргументы. После выполнения команды считываем данные из stdout и передаем их на вывод.

После написания основного функционала, остается обернуть запуск этого приложения внутри Java-агента. Пример кода представлен в листинге 8.

```

public class Agent {
    private static final Logger log = LoggerFactory.getLogger(Agent.class);
    private static final AtomicBoolean started = new AtomicBoolean(false);

    public static void premain(String agentArgs, Instrumentation inst) {
        runAgent();
    }

    public static void runAgent() {
        if (!started.compareAndSet(false, true)) {
            return;
        }
        try {
            String pid = new java.io.File("/proc/self").getCanonicalFile().getName();
            Process = Runtime.getRuntime().exec(new String[]{" /opt/runtime-diagnostic/agent-
jvm", String.valueOf(pid)}, null);
            listenProcessOutput(process.getInputStream());
            listenProcessOutput(process.getErrorStream());
            Runtime.getRuntime().addShutdownHook(new Thread(process::destroy));
        } catch (IOException e) {
            log.error("Agent startup failed", e);
        }
    }

    private static void listenProcessOutput(InputStream processOutput) {
        Thread = new Thread() -> {
            StringBuilder currentLine = new StringBuilder();
            try (BufferedReader is = new BufferedReader(new InputStreamReader(processOutput,
StandardCharsets.UTF_8))) {
                while (true) {
                    String line = is.readLine();
                    if (line == null) {
                        log.info(currentLine.toString());
                        break;
                    }
                    if (line.equals("\0")) {
                        log.info(currentLine.toString());
                        currentLine = new StringBuilder();
                    } else {
                        currentLine.append(line);
                    }
                }
            } catch (IOException e) {
                log.warn("Agent log init failed", e);
            }
        });
        thread.setDaemon(true);
        thread.start();
    }
}

```

Листинг 8 — Реализация агента

Реализация агента отличается от разработки обычного Java-приложения, где входной точкой является метод `main`. Началом работы же агента является метод `premain`, который выполняется до выполнения основного метода приложения. В этом методе, вызывается метод `runAgent`, который инициализирует написанный на Go сервис.

В методе `runAgent` сначала нужно убедиться, что агент ещё не запущен. Далее, чтобы иметь возможность собирать артефакты с приложения с помощью `jcmd`, необходимо получить его PID. После всех подготовительных шагов создаётся новый процесс, запускающий приложение, написанное на Go.

Важно помнить, что, если приложение завершит свою работу, JVM завершится, и следовательно, перестанет работать и приложение, и агент, но не сервис, написанный на Go. Чтобы этого избежать, добавляем событие `addShutdownHook`, которое срабатывает при завершении работы JVM и уничтожает процесс с приложением на Go.

Также необходимо учитывать логирование. Логи агента могут идти вместе с логами основного приложения, тогда как логи приложения на Go записываются в его собственные `stdout` и `stderr`. Поэтому создаются два слушателя, которые обрабатывают вывод и ошибки от Go-приложения и переносят их в логи основного приложения.

Чтобы запустить агента вместе с приложением, необходимо при запуске добавить флаг `-javaagent` и через двоеточие указать путь до агента. Например:

```
java -jar app.jar -javaagent:/home/agent.jar
```

3.3 Разработка доставщика команд

Начать разработку микросервисов было решено с доставщика команд. Доставщика команд отвечает за получение запросов на запуск сбора различных артефактов и их сохранение. Для передачи команд агенту было принято решение регулярно проверять, жив ли агент. Для этого используется синхронный подход, когда агент ходит на API доставщика. В случае асинхронного общения мы бы не смогли узнать это в реальном времени.

Для этого агент раз в минуту обращается к доставщику команд, чтобы узнать, есть ли команды для выполнения. Это позволяет не только передавать команды, но и проверять, что агент всё ещё активен и функционирует. Таким образом, мы можем предоставлять клиентам информацию только о работающих агентах, так как при создании команды клиент должен явно указать, на каком агенте она должна быть выполнена.

Чтобы снизить нагрузку на базу данных из-за частых запросов агентов, было решено создать хранилище для хранения необходимой информации о командах. Это позволит избежать постоянных обращений к базе данных. Реализация хранилища представлена в листинге 9 и 10.

```
private final ConcurrentHashMap<ParentId, HashSet<D>> index;

public List<D> get(ParentId id, long since) {
    lock.readLock().lock();
    try {
        var commands = index.get(id);
        if (commands != null) {
            return commands.stream()
                .filter(c -> c.createdAt() > since)
                .sorted(Comparator.comparing(Data::createdAt))
                .toList();
        }
    } finally {
        lock.readLock().unlock();
    }

    return List.of();
}
```

Листинг 9 — Получение команд из хранилища

В методе `get` реализуется потокобезопасное получение данных из `ConcurrentHashMap`. Это сделано для того, чтобы избежать небезопасного общего доступа к данным.

`ConcurrentHashMap` выбрана из-за ее потокобезопасности и высокой производительности. Она позволяет множеству потоков одновременно читать и добавлять данные без необходимости полной блокировки коллекции, что обеспечивает эффективную работу с многопоточностью.

```

public void put(ParentId id, D data) {
    lock.writeLock().lock();
    try {
        if (index.computeIfAbsent(id, (_id) -> new HashSet<>()).add(data)) {
            var sink = this.sinks.remove(id);
            if (sink != null) {
                sink.complete(List.of(data));
            }

            if (clearCountDown.decrementAndGet() == 0) {
                clearCountDown.set(CLEAR_COUNT_DOWN);
                var current = this.index.entrySet().stream().toList();
                for (Map.Entry<ParentId, HashSet<D>> entry : current) {
                    var commandsToRemove = entry.getValue().stream()
                        .filter(c -> c.createdAt() <= System.currentTimeMillis() -
TimeUnit.MINUTES.toMillis(10))
                        .toList();
                    for (D cmd : commandsToRemove) {
                        entry.getValue().remove(cmd);
                    }
                    if (entry.getValue().isEmpty()) {
                        this.index.remove(entry.getKey());
                    }
                }
            }
        }
    } finally {
        lock.writeLock().unlock();
    }
}

```

Листинг 10 — Сохранение команд в хранилище

Метод `put` реализует потокобезопасное сохранение данных. Кроме того, он проверяет необходимость очистки устаревших данных и удаляет их по истечению срока хранения, чтобы избежать переполнения памяти мертвыми данными.

Помимо работы с командами, сервис также реализует функционал регистрации агентов в базе данных и сохранения всей важной информации о них. Это позволяет клиенту получать список всех агентов, подключенных к его приложениям.

3.4 Разработка сборщика артефактов

Сборщик артефактов получает готовые артефакты от агента, выполняет необходимые манипуляции и сохраняет их в S3. Важным моментом является

решение архивировать артефакты для сокращения времени передачи между сервисами. Архивирование всех артефактов осуществляется на лету на стороне агента, и одновременно они отправляются на сборщика. Код обработки готового артефакта представлен в листинге 11.

```
public UploadControllerResponse upload(
    InputStream body,
    @Header("COMMAND-ID") @Nullable String commandId,
    @Header("TENANT") String tenant,
    @Header("AGENT-NAME") String agentName,
    @Header("UPLOAD-KIND") String kind,
    @Header("Content-Encoding") @Nullable String contentEncoding,
    ) throws IOException {
    try (body) {
        var kindValue = parseArtifactKind(kind);
        var stream = contentEncoding != null ? body : new DirectGzipInputStream(body);
        var agent = this.agentsRepository.findAgent(agentName);
        if (agent == null) {
            throw HttpServletResponseException.of(500, "Unknown agent: " + agentName);
        }
        var appName = Objects.requireNonNullElse(agent.appName(), "UNKNOWN");

        contentEncoding = contentEncoding != null ? contentEncoding : "gzip";
        var suffix = switch (contentEncoding) {
            case "gzip" -> ".gz";
            default -> "";
        };

        var key = tenant + '-' + agentName + '-' +
            (commandId != null ? commandId : LocalDateTime.now(clock)) +
            switch (kindValue) {
                case jvm_thread_dump -> "-thread_dump.txt";
                case jvm_heap_dump -> "-heap_dump.hprof";
                case oom_java_heap_dump -> "-oom_jvm_heap_dump.hprof";
            };

        var s3Artifact = s3Service.store(stream, key + suffix, contentEncoding);
        var artefactId = UUID.randomUUID().toString();
        var url = "/api/v2/" + tenant + "-" + artefactId + "/download";

        return new UploadControllerResponse(url);
    }
}
```

Листинг 11 — Сохранение артефакта в s3

Данный код принимает заархивированный артефакт в виде потока данных, который затем обрабатывается и загружается в S3. После этого

формируется URL, по которому пользователь сможет скачать артефакт через API. Пользователю останется только разархивировать его.

Как видно из кода, в данном методе не происходит сохранения профиля. Это связано с тем, что профилирование выполняется на протяжении заданного времени, и отправка такого большого файла может значительно ухудшить производительность сервисов. Чтобы решить эту проблему, было решено отправлять профиль по частям.

Когда поступает часть профиля, она сразу сохраняется в S3. Чтобы пользователь мог получить полностью упакованный профиль после завершения профилирования, при запросе профиля происходит объединение всех частей. Поиск частей осуществляется по их одинаковому префиксу.

Помимо сохранения артефактов, сборщик также предоставляет API для их скачивания. Поскольку артефакты хранятся в заархивированном виде, важно отметить, что современные браузеры умеют распаковывать архивы. Однако разархивация не всегда происходит корректно, из-за чего файлы могут оказаться повреждёнными.

Чтобы избежать этой проблемы, было решено отслеживать откуда происходит скачивания. Если скачивание происходит через браузер, Content-Encoding устанавливается в значение identity. Для определения того, что скачивание происходит именно из браузера, проверяется заголовок User-Agent. Во всех современных браузерах, таких как Safari, Google Chrome, Mozilla Firefox и других, заголовок User-Agent хранит в себе значение Mozilla.

Когда приходит запрос на скачивание, производится проверка на наличие Mozilla в заголовке User-Agent, и, при необходимости, Content-Encoding устанавливается в identity.

3.5 Разработка менеджера

Менеджер отвечает за запрос квот для S3 и очистку старых артефактов, чтобы избежать переполнения S3 ненужными данными. Процесс очистки данных представлен в листинге 12.

```

@ScheduleWithCron(value = "0 0 1 * * ?")
private void clearArtifacts() {
    var monthBefore = LocalDate.now().minusMonths(1).asStartOfDay(ZoneId.of("UTC"));
    var artifacts = s3Service.getStoredArtifacts();
    var keysToDelete = new ArrayList<ObjectIdentifier>();
    artifacts.forEach((s3Object -> {
        if (s3Object.lastModified().isBefore(monthBefore.toInstant())) {
            keysToDelete.add(
                ObjectIdentifier.builder()
                    .key(s3Object.key())
                    .build()
            );
        }
    }));

    s3Service.remove(keysToDelete);
}

```

Листинг 12 — Удаление старых артефактов в s3

4 Тестирование

4.1 Интеграционное тестирование

Завершив разработку системы, наступает время тестирования. Решено начать с написания интеграционных тестов. Для этого наилучшим решением будет использование библиотеки TestContainers, которая позволяет поднимать все необходимые контейнеры непосредственно в коде, значительно упрощая процесс тестирования.

Было решено реализовать пять тест-кейсов:

1. Регистрация агента;
2. Создание команды запуска дампа;
3. Загрузка дампа;
4. Загрузка ООМ дампа;
5. Загрузка профиля.

Проверку остального функционала решено оставить для ручного тестирования. В листингах 13 и 14 представлены два примера тестов.

```
@Test
void registerAgent() throws Exception {
    var agentName = UUID.randomUUID().toString();

    // Регистрация агента с уникальным именем
    try (var rs = CommandDeliveryServiceActions.registerAgentWithName(agentName)) {
        // Проверка, что код ответа HTTP в диапазоне 200-299
        assertThat(rs.code()).isBetween(200, 299);
    }

    // Проверка, что агент был успешно добавлен в базу данных
    var agents = PostgresActions.getAgents().stream().filter(a ->
a.name().equals(agentName)).toList();
    assertThat(agents.size()).isEqualTo(1);
}
```

Листинг 13 — Тест на регистрацию агента

```
@Test
void createDump() throws Exception {
    var agentName = UUID.randomUUID().toString();

    // Регистрация агента с уникальным именем
    try (var rs = CommandDeliveryServiceActions.registerAgentWithName(agentName)) {
        assertThat(rs.code()).isBetween(200, 299);
    }
}
```

```

// Проверка, что агент был успешно добавлен в базу данных
var agents = PostgresActions.getAgents().stream().filter(a ->
a.name().equals(agentName)).toList();
assertThat(agents.size()).isEqualTo(1);

// Создание дампа для зарегистрированного агента
try (var rs = CommandDeliveryServiceActions.createDumpWithAgentId(agentName)) {
    assertThat(rs.code()).isBetween(200, 299);
}

// Проверка, что команда для создания дампа была добавлена в базу данных
var commands = PostgresActions.getAgentCommands(agentName);
assertThat(commands.size()).isEqualTo(1);
assertThat(commands.get(0).agentId()).isEqualTo(agents.get(0).id());
}

```

Листинг 14 — Тест на создание команды

4.2 Ручное тестирование

Основное тестирование было решено оставить на этап ручного тестирования. Перед началом тестирования была произведена интеграция со сторонним сервисом, а в качестве пользовательского интерфейса используется консоль. Запустив приложение вместе с агентом, пытаемся снять дампы памяти, профиль и трассировку стека с приложения. Результат работы представлен на рисунке 5, 6 и 7.

```

id.nurullaev@macbook-PQHCVP4FN7 ~ % dp rtd jvm dump -t rtd -a commands-delivery-service-6959fd4968-67k17
Dump command started with id 816caf45-645b-4cb2-896a-701a29a718e3
Writing dump to rtd-commands-delivery-service-6959fd4968-67k17-816caf45-645b-4cb2-896a-701a29a718e3-heap_dump.hprof

```

Рисунок 5 — Запуск сбора дампа памяти

```

id.nurullaev@macbook-PQHCVP4FN7 ~ % dp rtd jvm profile -t rtd -a commands-delivery-service-6959fd4968-67k17 -d 1m
Profile command started with id fa98c410-7cb4-4247-b1f0-6a800d007909
Writing profile to commands-delivery-service-6959fd4968-67k17-fa98c410-7cb4-4247-b1f0-6a800d007909-2024-05-16_23_17_57.jfr

```

Рисунок 6 — Запуск сбора профиля

```

id.nurullaev@macbook-PQHCVP4FN7 ~ % dp rtd jvm thread-dump -t rtd -a commands-delivery-service-6959fd4968-67k17
Stacktrace command started with id eb6d54c7-36a4-4619-b838-ccb9125ab9a3
Writing thread dump to rtd-commands-delivery-service-6959fd4968-67k17-eb6d54c7-36a4-4619-b838-ccb9125ab9a3-thread_dump.txt

```

Рисунок 7 — Запуск сбора трассировки стека

Как видно, результат работы положительный. После загрузки артефактов их можно открыть с помощью IntelliJ IDEA, поскольку эта среда разработки имеет встроенный анализатор. Это продемонстрировано на рисунке 8.

Class	Count	Shallow	Retained	Biggest Objects	GC Roots	Merged Paths	Summary	Packages
byte[]	7 703	350,46 kB	350,04 kB					
java.lang.String	7 627	183,05 kB	503,44 kB	> java.lang.Object[2136], GC Root: Global JNI				
java.lang.Object[]	870	91,42 kB	561,15 kB	> java.lang.System, GC Root: Sticky class				
java.lang.Class	500	36 kB	197,14 kB	> java.util.concurrent.ConcurrentHashMap				
char[]	6	33,02 kB	33,02 kB	> java.io.PrintStream				
int[]	466	32,11 kB	28,42 kB	> java.lang.Module				
java.util.HashMap\$Node	994	31,81 kB	52,9 kB	> jdk.internal.loader.ClassLoaders\$AppClassLoader, GC Root: Global JNI				
java.util.concurrent.ConcurrentHashMap\$Node	897	28,7 kB	33,94 kB	> sun.nio.cs.StandardCharsets, GC Root: Sticky class				
java.util.HashMap\$Node[]	257	23,7 kB	66,21 kB	> java.lang.module.ModuleDescriptor				
java.util.concurrent.ConcurrentHashMap\$Node[]	19	19,31 kB	54,9 kB	> java.lang.module.Configuration				
java.util.HashMap	259	12,43 kB	72,27 kB	> java.lang.module.Configuration				

Рисунок 8 — Анализ дампа памяти

Теперь необходимо проверить получение агентов, результат показан на рисунке 9.

```

"contents": [
  {
    "kind": "RuntimeAgent",
    "attributes": {
      "id": "6959fd4968-67kl7",
      "kind": "jvm",
      "version": "0.1.0",
      "name": "commands-delivery-service-6959fd4968-67kl7",
      "tenant": "rtd",
      "appName": "commands-delivery-service",
      "lastPoll": 1715901353438,
      "active": true
    }
  },
  {
    "kind": "RuntimeAgent",
    "attributes": {
      "id": "6959fd4968-qdl7n",
      "kind": "jvm",
      "version": "0.1.0",
      "name": "commands-delivery-service-6959fd4968-qdl7n",
      "tenant": "rtd",
      "appName": "commands-delivery-service",
      "lastPoll": 1715901325459,
      "active": true
    }
  },
  {
    "kind": "RuntimeAgent",
    "attributes": {
      "id": "6bf7c77c5c-bgjm6",
      "kind": "jvm",
      "version": "0.1.0",
      "name": "artefacts-collector-6bf7c77c5c-bgjm6",
      "tenant": "rtd",
      "appName": "artefacts-collector",
      "lastPoll": 1715901323050,
      "active": true
    }
  }
]

```

Рисунок 9 — Получение агентов

ЗАКЛЮЧЕНИЕ

Таким образом, в ходе выполнения выпускной квалификационной работы был разработан способ сбора артефактов приложений, соблюдая ограничения, наложенные стандартом PCI-DSS. Разработка данного приложения позволит пользователям улучшить управление памятью, выявить уязвимости и понять, где они могли допустить ошибки. На этом развитие сервиса не прекратится. В планах – добавить поддержку других языков программирования, разработать систему уведомлений о случаях ООМ и внедрить поддержку асинхронного профилирования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Документация к инструменту jcmd. – URL: <https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr006.html> (дата обращения 11.02.2024)
2. Документация к инструменту jstack. – URL: <https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstack.html> (дата обращения 11.02.2024)
3. Документация к инструменту jinfo. – URL: <https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jinfo.html> (дата обращения 11.02.2024)
4. Документация к инструменту jmap. – URL: <https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jmap.html> (дата обращения 11.02.2024)
5. Документация к инструменту авторизации OPA. – URL: <https://www.openpolicyagent.org/docs/latest/> (дата обращения 11.02.2024)
6. Документация к языку программирования Java. – URL: <https://docs.oracle.com/en/java> (дата обращения 11.02.2024)
7. Документация к языку программирования Go. – URL: <https://go.dev/doc> (дата обращения 11.02.2024)
8. Документация к фреймворку Kora. – URL: <https://kora-projects.github.io/kora-docs/ru> (дата обращения 11.02.2024)
9. Документация к реляционной базе данных PostgreSQL. – URL: <https://www.postgresql.org/docs> (дата обращения 11.02.2024)
10. Документация к брокеру сообщений Apache Kafka. – URL: <https://kafka.apache.org/documentation> (дата обращения 11.02.2024)