МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«ТЮМЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ НАУК
Кафедра программной и системной инженерии

РЕКОМЕНДОВАНО К ЗАЩИТЕ В ГЭК
И ПРОВЕРЕНО НА ОБЪЕМ
ЗАИМСТВОВАНИЯ
Заведующий кафедрой

д.н., профессор
_____А.Г. Ивашко
_____2018 г.

МАГИСТЕРСКАЯ ДИСЕРТАЦИЯ

Прогнозирование загрузки сервисной шины предприятия
09.04.03 Прикладная информатика
Магистерская программа «Прикладная информатика в экономике»

Выполнил работу Студент 2 курса очной формы обучения	 *(Подпись)*	Кабардинский Евгений Олегович
Руководитель работы д.н., профессор	 *(Подпись)*	Ивашко Александр Григорьевич
Рецензент к.т.н., ст. преподаватель	 *(Подпись)*	Григорьев Андрей Викторович

Реферат

Магистерская диссертация по теме «Прогнозирование загрузки Сервисной Шины Предприятия» содержит 83 страниц текстового документа, 4 части, 31 рисунка, 3 таблиц, 23 используемых источников.

Объектом исследования является Сервисная Шина Предприятия (ESB).

Целью работы является сокращение расходов на компьютерные мощности.

Достигнутые результаты:

- Выделены характеристики запросов к ESB
- Подобраны данные, предоставляемые ESB для потребителей данных в корпоративной сети.
- Разработана виртуальная лаборатория для сбора данных о потреблении ресурсов ESB при обработке запроса к web-сервису
- Обучение модели прогнозирования на основе данных эксперимента

Область применения – обученная модель прогнозирования пригодна для планирования закупки компьютерных мощностей. С помощью данной модели инженер способен рассчитать оптимальную конфигурацию для виртуального сервера облачного провайдера, что позволит избежать приобретения серверов с избыточной мощностью.

Оглавление

Введение

Интеграция информационных систем различного класса остается актуальной темой для многих компаний и государственных организаций, независимо от отрасли или сферы их деятельности. При выполнении интеграции ключевым вопросом можно назвать выбор оптимальной платформы, которая позволит обеспечить надежность и быстродействие при совместной работе нескольких систем. Одна из наиболее распространенных причин необходимости интеграции связана с тем, что в большинстве компаний уже автоматизировано выполнение отдельных блоков задач, в частности, с помощью ERP, CRM-систем. Перед лицом, принявшим решение интегрировать существующие корпоративные приложения и системы, встает вопрос о выборе оптимальной платформы разработки интеграционного решения.

Одним из ключевых этапов в жизненном цикле программного обеспечения является “Этап развертывания компонентов системы”. Существует несколько процессов выполняемых в рамках данного этапа [1]:

- оптимизация плана компонентной конфигурации соответственно имеющей компьютерной базе пользователя, ее свойств и топологии
- развертывание отдельных программных компонентов
- создание целевой компонентной конфигурации

Проблемным является этап “Оптимизации плана компонентной конфигурации” включающий в себя задачу по определению компьютерных мощностей для программного обеспечения [1].

Задача планирования производственных мощностей имеет несколько возможных методов решения [2]:

- Метод чернового планирования мощностей
- Модуль планирования требуемых мощностей

- Кадровое обеспечение производственной линии
- Потребности в кадровом обеспечении гибкой рабочей ячейки

В данной работе мы рассмотрим планирования компьютерных мощностей для решения, разработанного на базе Сервисной Шины Предприятия с использование метода “Метод чернового планирования мощностей”.

Метод чернового планирования мощностей обычно применяется вручную или с помощью простой программы табличных вычислений. Он может использоваться для определения требуемых мощностей в долгосрочном и среднесрочном периодах. План-график мощностей составляется, включая текущие потребности в мощностях и прогнозируемые, на месяц вперед, чтобы получить необходимые ресурсы.

В данной работе мы рассмотрим способ получения показателей потребления ресурсов Сервисной Шиной Данных, а также время отклика на атомарный запрос, что позволит нам в дальнейшем использовать эти данные для решения задачи “Планирования производственных мощностей” методом “Метод чернового планирования мощностей”.

Целью работы является сокращение расходов на компьютерные мощности. Для достижения данной цели были выдвинуты следующие задачи:

- Разработка плана эксперимента Разработка типового решения на базе Сервисной Шины Предприятия пригодного для сбора данных об используемых им ресурсах
- Разработка виртуальной лаборатории, способной автоматически проводить серии экспериментов
- Сбор данных и их корреляционный анализ
- Обучение модели прогнозирования на основе данных этапа “Сбор данных и их корреляционный анализ”

1. Сервисная Шина Предприятия

Сервисная шина предприятия (англ. enterprise service bus, ESB) — связующее программное обеспечение, обеспечивающее централизованный и унифицированный событийно-ориентированный обмен сообщениями между различными информационными системами на принципах сервис-ориентированной архитектуры. Понятие введено в начале 2000-х годов специалистами подразделения Progress Software — Sonic, разрабатывающие MOM-продукт SonicMQ.

Основной принцип сервисной шины — концентрация обмена сообщениями между различными системами через единую точку, в которой, при необходимости, обеспечивается транзакционный контроль, преобразование данных, сохранность сообщений. Все настройки обработки и передачи сообщений предполагаются также сконцентрированными в единой точке, и формируются в терминах служб, таким образом, при замене какой-либо информационной системы, подключенной к шине, нет необходимости в перенастройке остальных систем.

Наименование подобрано по аналогии с системной шиной компьютера, позволяющей подключать несколько устройств и передавать данные между ними по одному набору проводников.

Интеграционное решение - приложение, разработанное на основе сервисной шины предприятия для интеграции систем предприятия.

1.1 Подходы к интеграции данных

Существует несколько основных подходов к интеграции данных в организациях[3]:

- Консолидация данных
- Федерализация данных
- Распространение данных

- Гибридный подход
- Сервисный подход

В соответствии с классификацией К. Диттриха [3], интеграцию данных возможно производить на нескольких уровнях архитектуры ИС организации (Рисунок 1):

Рисунок 1 Классификация типов интеграции

Сервисная шина данных позволяет реализовать интеграцию данных на следующих уровнях[3]:

- Common Data Storage (Общие системы хранения)
- Uniform Data Access (Унифицированный доступ к данным)
- Integration by Middleware (Интеграция средствами ПО промежуточного слоя)

В данной работе будет рассмотрено прогнозирование загрузки интеграционных приложений, решающих задачу “Uniform Data Access”, что накладывает дополнительные ограничения для интеграционного приложения: интерфейсы взаимодействия приложения должны быть доступны как во внутренней корпоративной сети, так и из сети интернет. Исходя из этого факта, все взаимодействия с интеграционным приложением должны

осуществляться с помощью web-сервисов или с помощью некоторых сетевых протоколов.

1.2 Микросервисы

Микросервисы и связанные с ними архитектуры набирают обороты на предприятиях всех размеров. Тем не менее, многие ИТ-специалисты все еще находятся в неведении относительно того, какие микросервисы действительно существуют и что представляет собой инфраструктура микросервисы для развития ИТ-решений на современных предприятиях. Для DevOps, микросервисы, также оказывают большое влияние. Гибкий цикл разработки объединяет то, что было двумя отдельными ИТ-подразделениями, разработками и операциями, в общепринятую идеологию проекта.

На очень высоком уровне микросервисы могут быть концептуализированы как новый способ создания корпоративных приложений, где приложения разбиваются на более мелкие независимые службы, которые не зависят от конкретного языка кодирования. Другими словами, команды разработчиков могут использовать языковые инструменты, с которыми им наиболее удобно работать, и по-прежнему добиваться синергии в процессе разработки приложений. Используя идеологию микросервисов, большие сложные приложения могут быть разделены на более мелкие строительные блоки исполняемых файлов, которые при перекомпоновке предлагают все функциональные возможности крупномасштабного, очень сложного приложения.

Принятие микросервисов позволяет организациям достичь большей гибкости и реализовать более низкие издержки благодаря пере использованию того, что представляет собой микросервис. Благодаря большому количеству разработчиков поддерживающих технологий и идеологий микросервисной архитектуры, концепция архитектур на основе микросервисов превратилась в

реальность для предприятий любого размера, что привносит гибкость и эффективность в разработку и развертывание приложений, так быстро, как это считалось невозможным.

Данные концепции и технологии включают в себя повышенную доступность контейнеров, инструментов и процессов. Объединяя их, создается среда разработки, которая больше фокусируется на сотрудничестве, чем на управлении кодом, что приводит к повышению производительности.

1.3 Микросервисная архитектура

Архитектурный стиль микросервисов использует идеологию разработки единого приложения как набор небольших, узко ориентированных, независимо развертываемых сервисов. Каждый микросервис работает в своем собственном процессе и несет в себе достаточно простую логику, часто обладает API-интерфейсом через протокол HTTP.Эти службы инкапсулируются для конкретных бизнес-возможностей и развертываются независимо, с использованием полностью автоматизированного механизма.

Микросервисы способствуют продвижение облачных решений, где несколько приложений могут управляться с помощью общего набора сервисов. Так же, архитектура на основе микросервисов оказывается полезной для реализации публичных, частных и гибридных облаков, где можно использовать концепцию использования набора небольших независимых сервисов, с использованием легкого интерфейса, такого как API RESTful.

Возможно, концепция SOA (Service Oriented Architecture), в которой основное внимание уделялось обнаружению сервисов и использованию анонимных сервисов на основе UDDI (Universal Description, Discovery and Integration), по-прежнему прослеживается в микросервисах архитектуры. Однако, микросервисы идут на несколько шагов вперед, используя API RESTful, вместе с виртуализованными платформами и контейнерами, чтобы

создать интеграцию гибридных сервисов, которая заменяет монолитные однокомпонентные базовые приложения.

Однако микросервисы, основанные на принципах SOA, в настоящее время являются реальностью, пришедшей с внедрением новых технологий, которые составляют три основных строительных блока архитектуры микросервисов:

- Контейнеры. Программные контейнеры создали стандартизованный фрейм для всех сервисов, абстрагировав основной код ОС с базовым оборудованием. Стандартизация, предлагаемая контейнерами, устраняет то, что когда-то было болезненным процессом интеграции в гетерогенном мире инфраструктуры. Благодаря таким инновациям, как Docker, контейнеры революционизировали. Это так же повлияло на то, как разработчики создают и развертывают свои приложения.
- API. внедрение и расширение возможностей, предлагаемых API, создало надежный и стандартизованный формат для взаимодействия между приложениями, службами и серверами. API-интерфейсы, REST (представление состояния передачи), в частности, являются ключевыми для архитектуры микросервисов: API RESTful разбивает транзакцию на создание серии небольших запросов, каждый из которых выполняет определенную часть транзакции. Эта модульность предоставляет разработчикам большую гибкость при разработке легких API-интерфейсов, которые более подходят для приложений с поддержкой браузера.
- Масштабируемые облачные инфраструктуры. Открытые, частные и гибридные облачные инфраструктуры теперь способны доставлять ресурсы по требованию и могут эффективно масштабироваться для доставки услуг независимо от нагрузки.

Это придает эластичность микросервисам и, в свою очередь, делает их более гибкими и эффективными.

1.4 Архитектура ESB

Одна из основных задач при реализации SOA архитектуры в интеграционном приложении, это логическое разделение слоев приложения по обязанностям. Использование SOA архитектуры позволяет [7]:

- Уменьшить связанность компонентов
- Пере использовать компоненты
- Использовать интерфейс взаимодействия с удаленными системами, а не конкретную имплементацию

Разбиение интеграционного приложения на слои позволит систематизировать и структурировать решение, что очень важно для разработки и поддержки решения в будущем [7].

Для обеспечения высокой производительности и отказоустойчивости часто прибегают к использованию механизмов Load Balancing, а также использования выделенного ip для каждого сервиса ESB [7].

Выделяют три основных слоя на которые делится интеграционное решение:

- Experience (Channel) уровень
- Process (Enterprise) уровень
- System (Adapter) уровень

Данную архитектуру можно изобразить следующим образом (Рисунок 2):

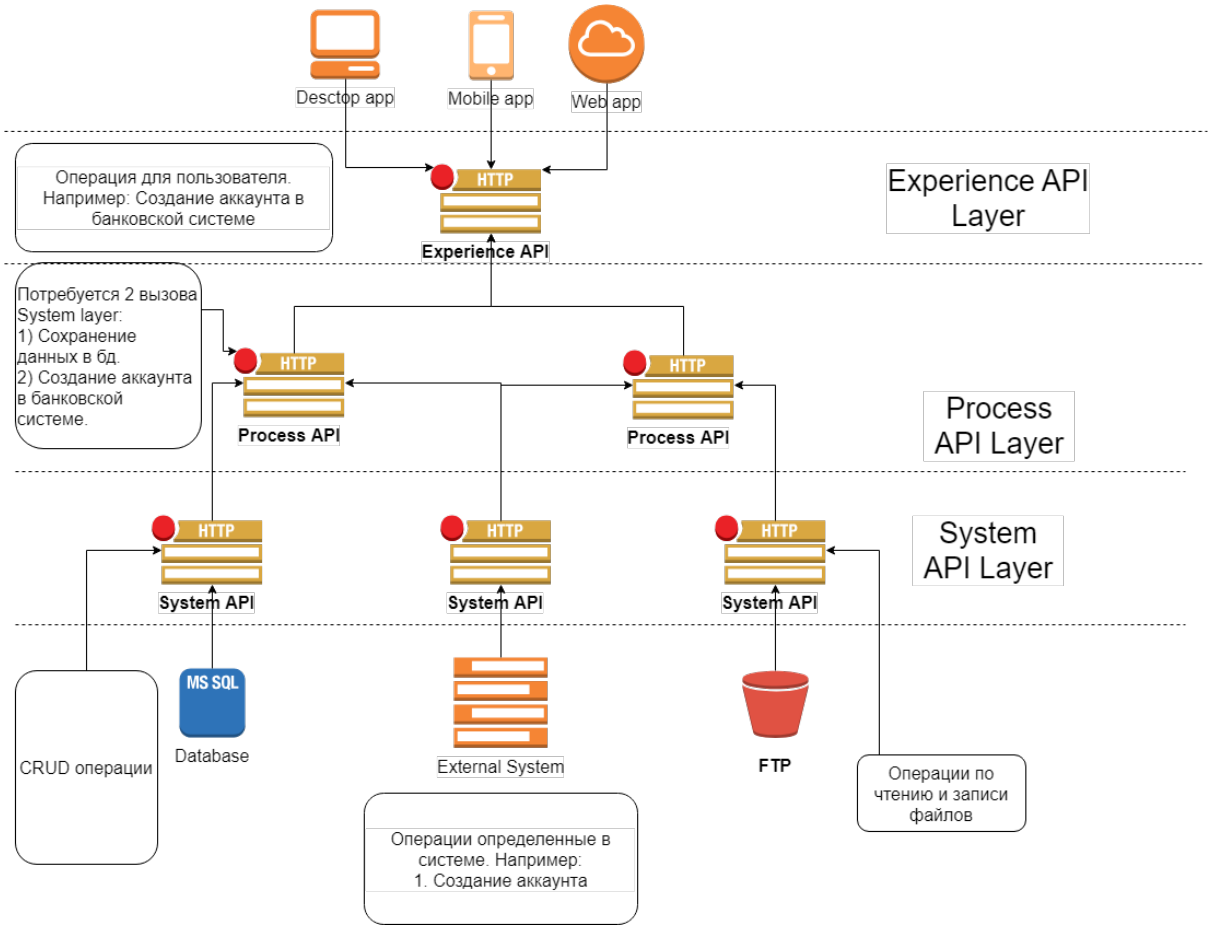

Рисунок 2 Трехуровневая архитектура приложения

1.5 Experience уровень

Уровень, на котором пользователи интеграционного решения могут получить доступ к сервисам, расположенным в данном решении. Основная задача уровня преобразовать запрос пользователя в канонический запрос интеграционного решения, а также преобразовать ответ решения в формат ответа с которым пользователь сможет работать.

1.6 Process уровень

Получает запросы от experience уровня. Уровень является посредником между Experience уровнем и уровнем System. Также может работать с слоями в корпоративной системе, но данные слои считаются как независимые.

1.7 System уровень

Получает запрос от Process уровня. Основной задачей является предоставление доступа к внешним системам.

1.8 Тест кейс

Тестовый случай (Test Case) - это артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части [8]

Enterprise Integration Patterns — шаблоны и стратегии интеграции корпоративных приложений с помощью механизмов обмена сообщениями. [9].

Исходя из того, что понятие ESB предполагает использование EIP в качестве строительных блоков интеграционном приложении [10], следует то, что потоки для разрабатываемого тест-кейса должны быть описаны в виде EIP диаграмм (Рисунок 3).

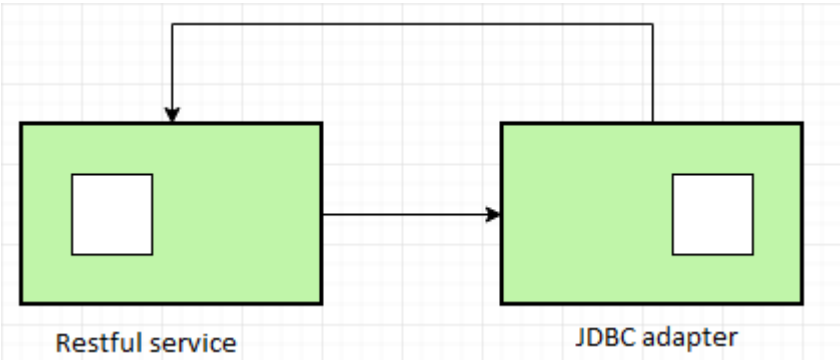

Рисунок 3 EIP диаграмма интеграционного потока

1.9 Интеграционное тестирование

Интеграционное тестирование — одна из фаз тестирования программного обеспечения, при которой отдельные программные модули объединяются и тестируются в группе. Обычно интеграционное тестирование проводится после модульного тестирования и предшествует системному

тестированию. Целью интеграционного тестирования является проверка соответствия проектируемых единиц функциональным, приёмным и требованиям надежности. Тестирование этих проектируемых единиц — объединения, множества или группы модулей — выполняется через их интерфейс, с использованием тестирования «чёрного ящика» [8].

Исходя из того факта, что каждый тест должен иметь свой интерфейс, был разработан интеграционный тест, для каждого теста, а также для каждого из управляющих интерфейсов. Интеграционные тесты были разработаны с помощью утилиты Postman, позволяющей отправлять произвольные http запросы. Каждый интеграционный тест включает в себя запрос к интерфейсу интеграционного приложения, а также проверку ответа на предмет наличие необходимых полей данных (Листинг кода 1).

```
1  tests["Status code is 200"] = responseCode.code === 200;
2
3  var jsonData = JSON.parse(responseBody);
4  tests["Is array"] = Array.isArray(jsonData)
5  tests["Not empty"] = jsonData.length > 0
6  tests["Has usedMemory"] = typeof jsonData[0].usedMemory === "number"
7  tests["Has systemCpuLoad"] = typeof jsonData[0].usedMemory === "number"
8  tests["Has systemTime"] = typeof jsonData[0].usedMemory === "number"
9  tests["Has stateNumber"] = typeof jsonData[0].usedMemory === "number"
```

Листинг кода 1 Валидация ответа от сервиса

Утилита также позволяет выполнить все описанные интеграционные тесты разом, что сокращает время интеграционного тестирования и полностью автоматизирует его (Рисунок 4).

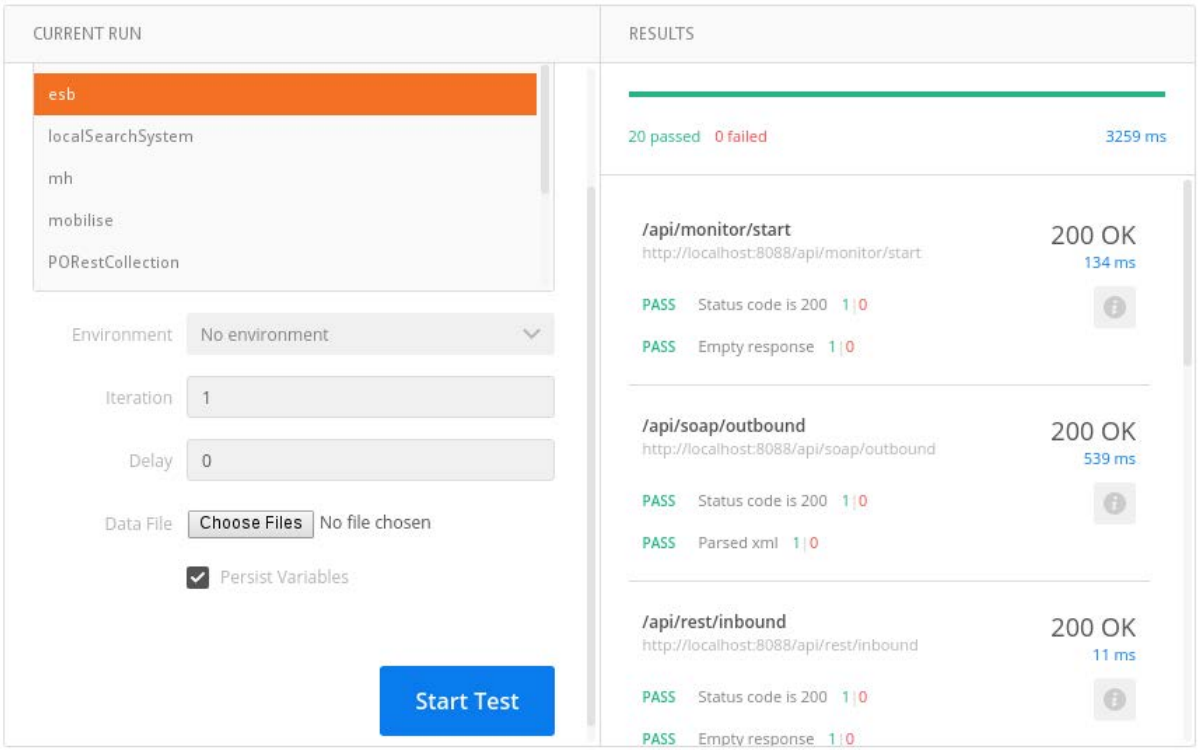

Рисунок 4 Инструмент автоматического интеграционного тестирования

1.10 Разработка сервисов источников данных

Тестирование сервисной шин данных предполагает тестирование реализации работы с различными протоколами в каждой из сервисных шин предприятия. Для тестирования реализаций протоколов в сервисной шине предприятия, необходимо предоставить сервисы провайдеры, позволяющие сервисной шине предприятия обращаться к данным сервисам, используя некоторый протокол. Были выделены следующие протоколы взаимодействия систем:

- TCP
- UDP
- HTTP
- FTP
- JDBC

Следует отметить, что с помощью протокола HTTP на данный момент реализуются следующие виды web-сервисов:

- Rest
- Soap

В итоге можно выделить 3 основных типов коммуникаций между системами:

- Протоколы базирующиеся на udp
- Протоколы базирующиеся на tcp
- Http web-сервисы
- FTP передача файлов
- JDBC доступ к базе данных

Для минимизации задержек на стороне сервисов заглушек, все сервисы предоставляют заранее подготовленный неизменяющийся ответ, а также исключены любые вычисления.

В данной работе были использованы только следующие типы источников данных:

- JDBC доступ к базе данных

JDBC (англ. Java DataBase Connectivity — соединение с базами данных на Java) — платформенно-независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД, реализованный в виде пакета java.sql, входящего в состав Java SE.

JDBC основан на концепции так называемых драйверов, позволяющих получать соединение с базой данных по специально описанному URL. Драйверы могут загружаться динамически (во время работы программы). Загрузившись, драйвер сам регистрирует себя и вызывается автоматически, когда программа требует URL, содержащий протокол, за который драйвер отвечает.

JDBC API содержит два основных типа интерфейсов: первый — для разработчиков приложений и второй (более низкого уровня) — для разработчиков драйверов.

Соединение с базой данных описывается классом, реализующим интерфейс java.sql.Connection. Имея соединение с базой данных, можно создавать объекты типа Statement, служащие для исполнения запросов к базе данных на языке SQL.

Существуют следующие виды типов Statement, различающихся по назначению:

- java.sql.Statement — Statement общего назначения;
- java.sql.PreparedStatement — Statement, служащий для выполнения запросов, содержащих подставляемые параметры (обозначаются символом '?' в теле запроса);
- java.sql.CallableStatement — Statement, предназначенный для вызова хранимых процедур.

Интерфейс java.sql.ResultSet позволяет легко обрабатывать результаты запроса.

Преимуществами JDBC считают:

- Лёгкость разработки: разработчик может не знать специфики базы данных, с которой работает;
- Код практически не меняется, если компания переходит на другую базу данных (количество изменений зависит исключительно от различий между диалектами SQL);
- Не нужно устанавливать громоздкую клиентскую программу;
- К любой базе можно подсоединиться через легко описываемый URL.

1.11Разработка окружения

Для быстрого развертывания клиентских приложений, интеграционного решения, базы данных в произвольной компьютерной сети было решено использовать платформу контейнеризации приложений. Платформа должна управлять виртуальными машинами, а также виртуальными сетями. Также платформа должна позволять использовать публичные репозитории образов систем, для обеспечения возможности использовать готовые, настроенные образы.

Исходя из требований, предъявляемых к платформе контейнеризации было решено использовать платформу контейнеризации linux приложений Docker.

Docker — программное обеспечение для автоматизации развёртывания и управления приложениями в среде виртуализации на уровне операционной системы. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть перенесён на любую Linux-систему с поддержкой cgroups в ядре, а также предоставляет среду по управлению контейнерами. Программное обеспечение функционирует в среде Linux с ядром, поддерживающим cgroups и изоляцию пространств имён (namespaces); существуют сборки только для платформы x86-64 [16].

Для экономии дискового пространства проект использует файловую систему Aufs с поддержкой технологии каскадно-объединённого монтирования: контейнеры используют образ базовой операционной системы, а изменения записываются в отдельную область. В состав программных средств входит демон — сервер контейнеров (запускается командой docker -d), клиентские средства, позволяющие из интерфейса командной строки управлять образами и контейнерами, а также API, позволяющий в стиле REST управлять контейнерами программно. Демон обеспечивает полную изоляцию запускаемых на узле контейнеров на уровне файловой системы (у каждого контейнера собственная корневая файловая система), на уровне процессов (процессы имеют доступ только к собственной файловой системе контейнера,

а ресурсы разделены средствами libcontainer), на уровне сети (каждый контейнер имеет доступ только к привязанному к нему сетевому пространству имён и соответствующим виртуальным сетевым интерфейсам).

Главной особенностью Docker является возможность расширять образы, используя встроенный декларативный язык создания образов.

Пример описания образа для http web сервиса, использующего публичный образ Tomcat:

```
FROM tomcat:9.0
COPY tomcat-users.xml /usr/local/tomcat/conf/tomcat-users.xml
COPY soap-service-1.0-SNAPSHOT.war /usr/local/tomcat/webapps/soap-service.war
```

Листинг кода 2 Dockerfile для soap-service приложения

Сборку данного образа можно произвести с помощью следующей команды:

docker build --tag=”soap-service”

Для запуска образа в отдельном контейнере потребуется следующая команда:

docker run --name soap-service -d soap-service

Все приложения заглушки были “обернуты” в соответствующие docker контейнеры. Для управления развертыванием окружения было принято решение использовать Docker Composer tool. Пример описания сервисов-заглушек:

Все приложения были “обернуты” в соответствующие docker контейнеры. Для управления развертыванием окружения было принято решение использовать Docker Composer tool. Docker compose контролирует процесс сборки и развертывания контейнеров, в зависимости от конфигурации, описанной в специализированном дескрипторе docker-compose.yml. Помимо запуска и сборки, инструмент позволяет задать перенаправление портов, задать переменные окружения, задать проброс директорий, задать зависимые контейнеры (Рисунок 5).

Рисунок 5 Привер docker-compose.yml файла

Docker compose контролирует процесс сборки и развертывания контейнеров, в зависимости от конфигурации, описанной в специализированном дескрипторе docker-compose.yml. Помимо запуска и сборки, инструмент позволяет задать перенаправление портов, задать переменные окружения, задать проброс директорий, задать зависимые контейнеры.

1.12Реализация интеграционного приложения

Рассмотрим несколько популярных продуктов реализующих принципы EIP:

- Apache Camel
- Spring integration
- Mule ESB
- WSO2 ESB

1.12.1 Apache Camel

Apache Camel — открытый кроссплатформенный java-фреймворк, который позволяет проводить интеграцию приложений в простой и понятной форме. Идеологически основан на Шаблонах Интеграции Корпоративных Приложений [11].

Особенности:

- Гибкая маршрутизация сообщений
- Более 70-ти различных компонентов для доступа к данным
- Не навязывается канонический формат данных на сообщения
- Маршруты описываются на Java DSL, XML DSL, Scala DSL
- Использование POJO-объектов возможно для любых целей, например, для трансформации сообщений
- Минимальные требования к конфигурации
- Автоматическая конвертация сообщений между различными форматами
- Легко может быть встроен в существующие приложения
- В составе идут инструменты для тестирования готового интеграционного решения
- Готов к размещению в OSGi-окружении

Пример маршрута, записанного с помощью Java DSL:

```
from("file:src/data?noop=true").
        choice().
        when(xpath("/person/city'London'")).to("file:target/messages/uk").
                otherwise().to("file:target/messages/others");
```

Spring Boot - платформа, позволяющая создавать полноценные, производственного класса Spring-приложения, про которые можно сказать - "просто запусти" [12]. Главной особенностью платформы является минификация Spring-конфигурации. Основные возможности:

- Создание полноценных Spring приложений

- Встроенный Tomcat или Jetty (не требуется установки WAR файлов)
- Обеспечивает 'начальные' POMs для упрощения Maven конфигурации
- Автоматическая конфигурация Spring когда это возможно
- Обеспечивает такими возможностями, как метрики, мониторинг состояниями и расширенная конфигурация
- Конфигурирование без генерации кода и без написания XML

Разработанное интеграционное приложение полностью реализует описанный выше тест кейс.

1.12.2 Spring integration

Использование Spring Framework стимулирует разработчиков к использованию интерфейсов в коде и внедрения зависимостей(DI), обеспечивая POJO-объекты необходимыми для выполнения поставленной задачи зависимостями. Spring Integration развивает эту концепцию на шаг дальше, где POJO-объекты связаны друг с другом с помощью парадигмы обмена сообщениями и отдельные компоненты могут быть не знать о существовании других компонентов в приложении. Такое приложение построено путем соединения мелких повторно используемых компонентов, чтобы сформировать высокий уровень функциональности. При тщательном проектировании, эти потоки могут быть модульными, а также повторно использованы на на более высоком уровене.

В дополнение к соединению вместе мелких компонентов, Spring Integration предоставляет широкий выбор адаптеров каналов и шлюзов для связи с внешними системами. Адаптеры каналов используются для односторонней интеграции (отправлять или получать); шлюзы используются для запроса/ответа сценариев (входящих и исходящих). Для получения полного списка адаптеров и шлюзов, обратитесь к справочной документации.

Spring XD основано на Spring Integration, где Spring Integration модули собраны в XD Stream

Возможности:

- Реализация большинства Enterprise Integration Patterns
  - Endpoint
  - Channel (Point-to-point and Publish/Subscribe)
  - Aggregator
  - Filter
  - Transformer
  - Control Bus
  - Другие...
- Интеграция с внешними системами
  - ReST/HTTP
  - FTP/SFTP
  - Twitter
  - WebServices (SOAP and ReST)
  - TCP/UDP
  - JMS
  - RabbitMQ
  - Email
  - Другие...
- Расширенная поддержка JMX
  - Представление компонентов фреймворка как MBeans
  - Адаптеры получают атрибуты из MBeans, запускают операции, отправляют/получают уведомления

Пример простого интеграционного потока:

```
<!-- Simple Service -->
<int:gateway id="simpleGateway"
    service-interface="foo.TempConverter"
    default-request-channel="simpleExpression" />
```

```
<int:service-activator id="expressionConverter"
    input-channel="simpleExpression"
    expression="(payload - 32) / 9 * 5"/>

<!-- Web Service -->

<int:gateway id="wsGateway" service-interface="foo.TempConverter"
    default-request-channel="viaWebService" />

<int:chain id="wsChain" input-channel="viaWebService">
    <int:transformer
        expression="'&lt;FahrenheitToCelsius
xmlns=''http://tempuri.org/''&gt;&lt;Fahrenheit&gt;XXX&lt;/Fahrenheit&gt;&
lt;/FahrenheitToCelsius&gt;'.replace('XXX', payload.toString())" />
    <int-ws:header-enricher>
        <int-ws:soap-action
value="http://tempuri.org/FahrenheitToCelsius"/>
    </int-ws:header-enricher>
    <int-ws:outbound-gateway
        uri="http://www.w3schools.com/webservices/tempconvert.asmx"/>
    <int-xml:xpath-transformer
        xpath-expression="/*[local-
name()='FahrenheitToCelsiusResponse']/*[local-
name()='FahrenheitToCelsiusResult']"/>
</int:chain>
```

Листинг кода 3 интеграционный поток в spring integration

1.12.3 Mule ESB

Mule ESB — это легковесная интеграционная платформа (сервисная шина предприятия - ESB), которая позволяет разработчику объединять различные информационные системы на основе принципов обмена сообщениями (message routing), сопоставления данных (data mapping), управления сообщениями (orchestration), надежности (контроль за обменом сообщениями), защиты (использование https и опциональных коннекторов) и масштабирования между узлами (коннекторами). Mule ESB является открытым программным обеспечением (CPAL-лицензия).

Пример интеграционных потоков, разработанных на платформе Mule ES:

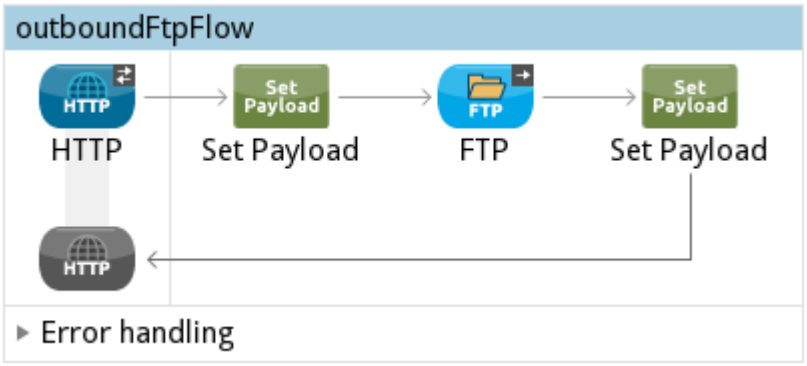

Рисунок 6 Поток для операции создания файла на ftp сервере

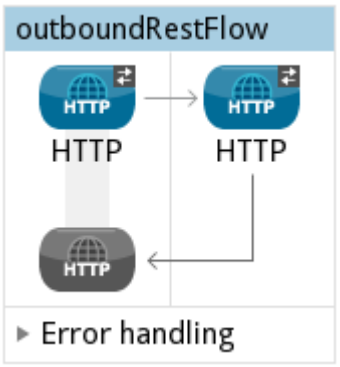

Рисунок 7 Реализация прокси запроса

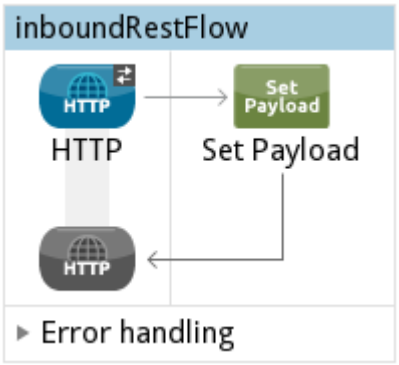

Рисунок 8 Статический ответ

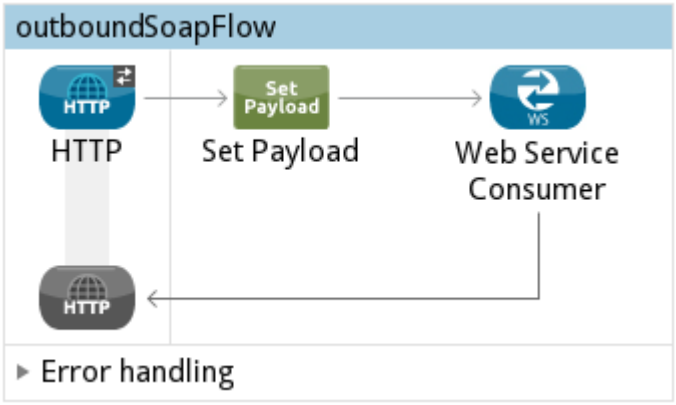

Рисунок 9 Прокси Soap запрос без пост обработки ответа

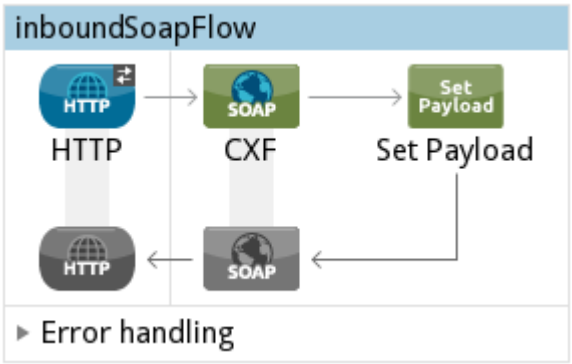

Рисунок 10 Сервис Soap адаптер с пост обработкой ответа

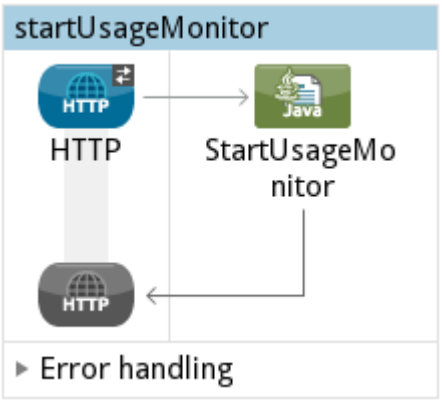

Рисунок 11 Вызов java класса для получения ответа

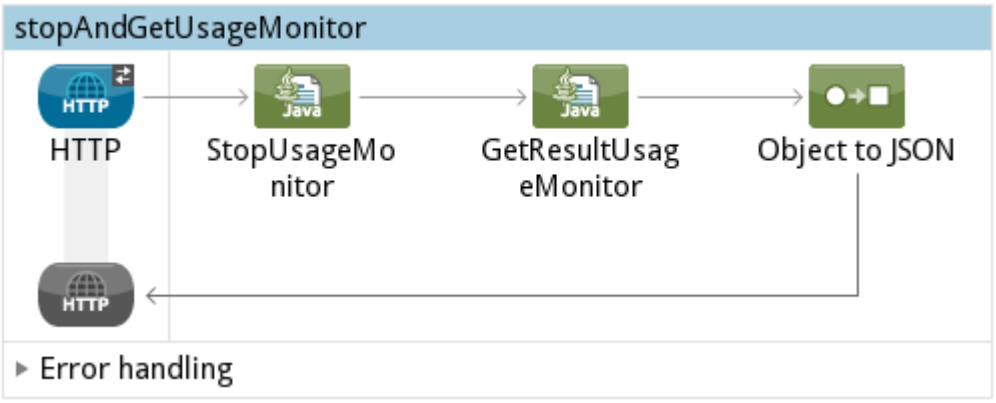

Рисунок 12 Цепочка java вызовов с поочередной передачей интеграционного сообщения между java классами

1.12.4WSO2 ESB

Интеграционная шина WSO2 ESB - легкое и быстрое решение в области интеграции информационных систем. Обладая поддержкой всех протоколов, имея широкий набор коннекторов к различным системам, данный продукт идеально подходит для задачи интеграции

Ключевые возможности WSO2 ESB:

- Более 100 коннекторов позволяют легко подключить наиболее популярные сервисы, например такие как Salesforce, Paypal, LinkedIn, Twitter, JIRA;
- Поддерживаемые транспорты: HTTP, HTTPS, POP, IMAP, SMTP, JMS, AMQP, FIX, TCP, UDP, FTPS, SFTP, CIFS, MLLP и SMS;
- Поддерживаемые форматы и протоколы: JSON, XML, SOAP 1.1, SOAP 1.2, WS-*, HTML, EDI, HL7, OAGIS, Hessian, Text, JPEG, MP4, all binary formats and CORBA/IIOP;
- Поддерживаемые системы: SAP BAPI & IDoc, PeopleSoft, MS Navision, IBM WebSphere MQ, Oracle AQ и MSMQ
- Реализация всех EIP (Enterprise Intergration Patterns);
- Преобразование данных: XSLT 1.0/2.0, XPath, XQuery и Smooks;
- Кластеризация и балансировка нагрузки для обработки большого потока запросов;

- Широкие возможности по авторизации и аутентификации запросов и данных: WS-Security, LDAP, Kerberos, OpenID, SAML, XACML;
- Мощная панель администрирования на базе WSO2 Carbon
- Мониторинг, логирование.

Пример Rest API реализованного с помощью WSO2:

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
    <!-- You can add any flat sequences, endpoints, etc.. to this synapse.xml
file if you do *not* want to keep the artifacts in several files -->
  <api name="StockQuoteAPI" context="/stockquote">
     <resource uri-template="/view/{symbol}" methods="GET">
        <inSequence>
           <payloadFactory>
             <format>
                <m0:getQuote xmlns:m0="http://services.samples">
                   <m0:request>
                     <m0:symbol>$1</m0:symbol>
                   </m0:request>
                </m0:getQuote>
             </format>
              <args>
                <arg expression="get-property('uri.var.symbol')"/>
              </args>
           </payloadFactory>
           <header name="Action" value="urn:getQuote"/>
           <send>
             <endpoint>
                <address uri="http://localhost:9000/services/SimpleStockQuoteServ
ice" format="soap11"/>
             </endpoint>
           </send>
        </inSequence>
        <outSequence>
           <send/>
        </outSequence>
     </resource>
     <resource url-pattern="/order/*" methods="POST">
        <inSequence>
           <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
           <property name="OUT_ONLY" value="true"/>
           <send>
              <endpoint>
```

```
                    <address uri="http://localhost:9000/services/SimpleStockQuoteSe
rvice" format="soap11"/>
                </endpoint>
            </send>
        </inSequence>
    </resource>
  </api>
</definitions>
```

Листинг кода 4 интеграционный поток в WSO2 ESB

Для разработки интеграционного решения в данной работе выступает связка технологий: Spring Boot + Apache Camel.

1.13Автоматическая отправка запросов

Основной задачей ESB является предоставление единого унифицированного доступа к данным предприятия посредством web-сервисов [3]. Исходя из этого факта было принято решение о разработке клиентского приложения, задачей которого является вызов сервисов, предоставляемых ESB. Предполагается, что в одном тестовом окружении будет запущено несколько копий клиентского приложения, каждая из которых должна работать на отдельной физической машине. Главным требованием к клиентскому приложению является его способность последовательно вызывать определенный web-сервис в определенном временном промежутке. Для обеспечения синхронного вызова сервисов ESB все запущенные копии клиентских приложений должны управляться мастер-приложением (Рисунок 13).

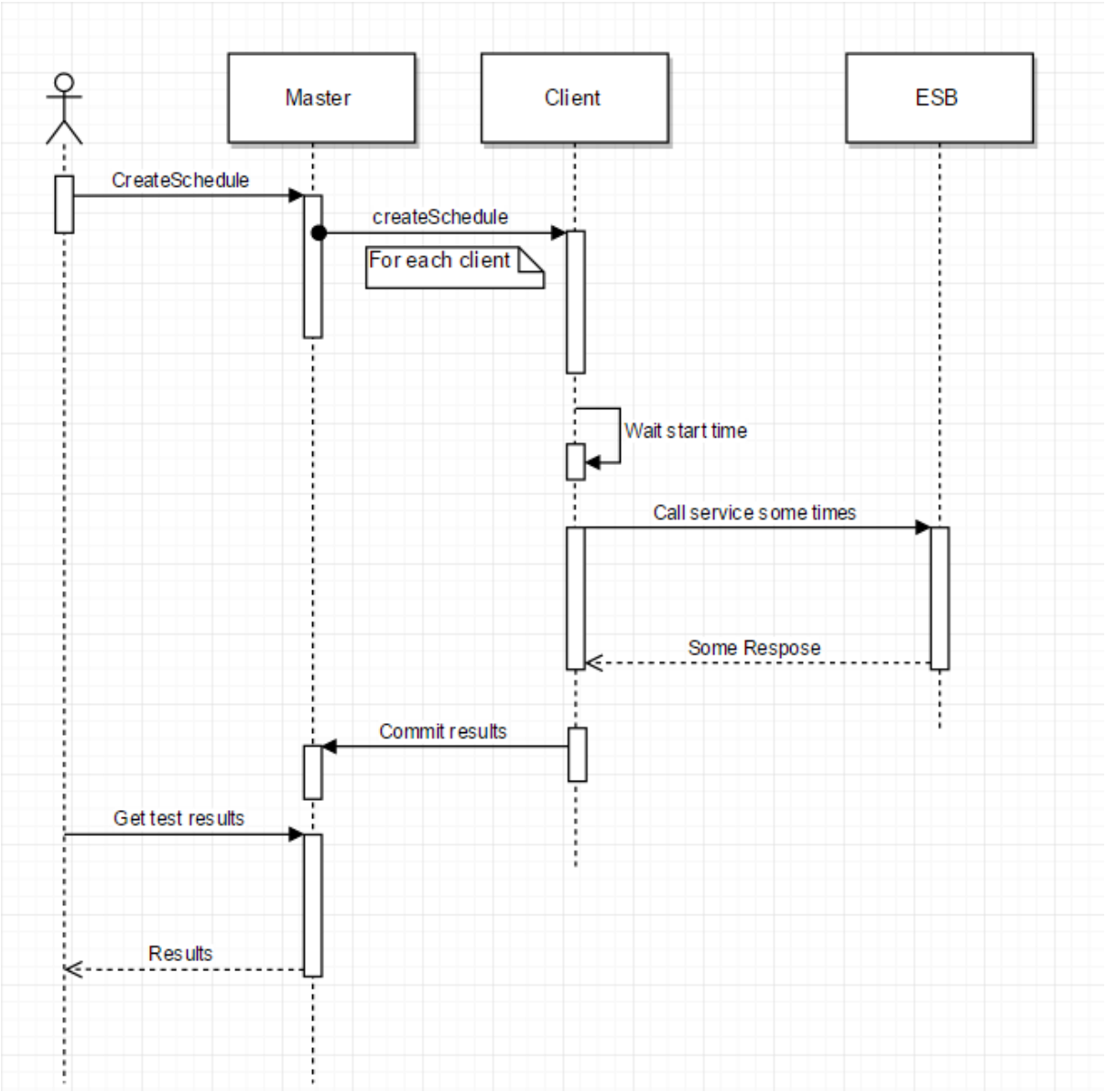

Рисунок 13 Схема взаимодействия тестирующей системы

Все параметры использования ресурсов интеграционным приложением измеряются во время отправки запроса.

2. Java Virtual Machine

Java Virtual Machine это среда для запуска Java приложений. При запуске Java программы вызывается метод main, который реализован в java коде. JVM это часть JRE (Java Runtime Engine).

JRE это среда для запуска Java приложений, не путать с JDK. JDK (Java Development Kit - это набор инструментов, позволяющий разрабатывать приложения на Java, включающий в себя как JRE так компилятор Java кода и библиотеки для разработки Java приложений).

Java приложения называются WORA (Write Once Run Everywhere) приложениями, то есть буквально напиши один раз и запускай везде. Это означает что разработчик может создать Java код на одной операционной системе и запустить его в другой системе, в которой установлена JRE без каких-либо дополнительных настроек. Это все возможно потому что это JVM.

Когда мы компилируем .java файл, создается *.class файл (java byte code) с таким же названием, этот файл генерируется Java компилятором.

1.14JVM JIT оптимизация

Исходный компилятор Java (javac) читает исходные файлы Java и компилирует их в файлы классов. В отличие от многих других компиляторов, таких как gcc или ghci, он не создает собственный исполняемый код для данной целевой архитектуры, но создает байт-код.

Байт-код состоит из набора команд, где все коды операций представлены в одном байте. Инструкции выполняются на виртуальной машине. Благодаря этому байт-код является агностиком архитектуры, но он также намного менее эффективен сам по себе, чем оптимизированный нативный код.

Другое отличие от компиляторов, упомянутых выше, состоит в том, что компиляция байт-кода не требует значительной оптимизации всего лишь

нескольких, например, вложенных констант. (Это может стать для вас проблемой, если вы решитесь напрямую изменять файлы классов.)

Это означает, что полученный байт-код имеет сходные характеристики с исходным кодом. Это сходство открывает возможность декомпиляции приложений с легкостью и упрощает манипуляции с байт-кодом, что отлично, например, если вы решили собирать назначения переменных для неудачных тестов.

Ранние версии JVM не имели каких-либо оптимизаций и работали с байт-кодом, напрямую интерпретируя коды операций. Чтобы сделать программы более эффективными, технология HotSpot была представлена в JDK 1.3 для обеспечения компиляции «Just-In-Time», позволяющей среде выполнения выполнять оптимизированный код, а не медленно интерпретировать инструкции байт-кода.

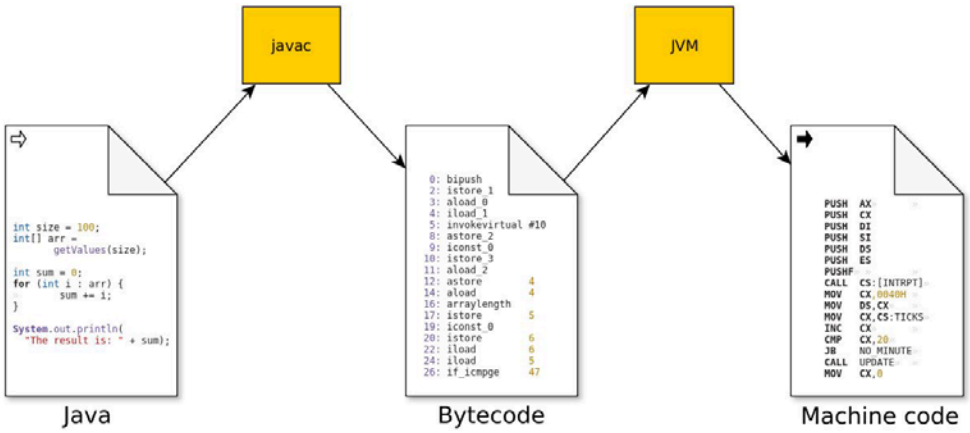

Рисунок 14 Схема компиляции кода «на лету»

JVM во время выполнения программы, внимательно наблюдает за кодом, когда он выполняется на виртуальной машине. Например, он анализирует, какие части кода выполняются часто, чтобы решить, что стоит оптимизировать, а также анализирует, как выполняется код для определения того, какие оптимизации могут быть применены.

JIT Compiler создает оптимизированный код, адаптированный к целевой архитектуре, который заменяет интерпретируемую версию «на лету».

Компиляция происходит в фоновом потоке, поэтому она фактически не прерывает выполнение программы.

Для некоторых оптимизаций требуется более подробный профиль, для некоторых из них нет, поэтому некоторые оптимизации могут применяться раньше других, а для некоторых из них потребуется немного усилий, если приложение будет работать в течение длительного времени.

1.15Типы компиляторов JVM

В JVM есть два компилятора: клиент (c1) и сервер (c2).

Компилятор клиента выполняет более простые оптимизации, поэтому для анализа текущего кода требуется меньше времени. Он лучше всего подходит для относительно коротких клиентских приложений, обеспечивая более быстрый запуск, но меньшую оптимизацию. Компилятор сервера нацелен на приложения, которые работают в течение длительного времени. Он собирает больше данных для профилирования и запускается медленнее, но предлагает более сложные методы оптимизации.

С Java 6 нужно было выбирать между c1 и c2 компиляторами. Начиная с Java 7 существует возможность использовать многоуровневую компиляцию, а с Java 8 это поведение JVM по умолчанию. В этом подходе используются как c1, так и c2-компиляторы.

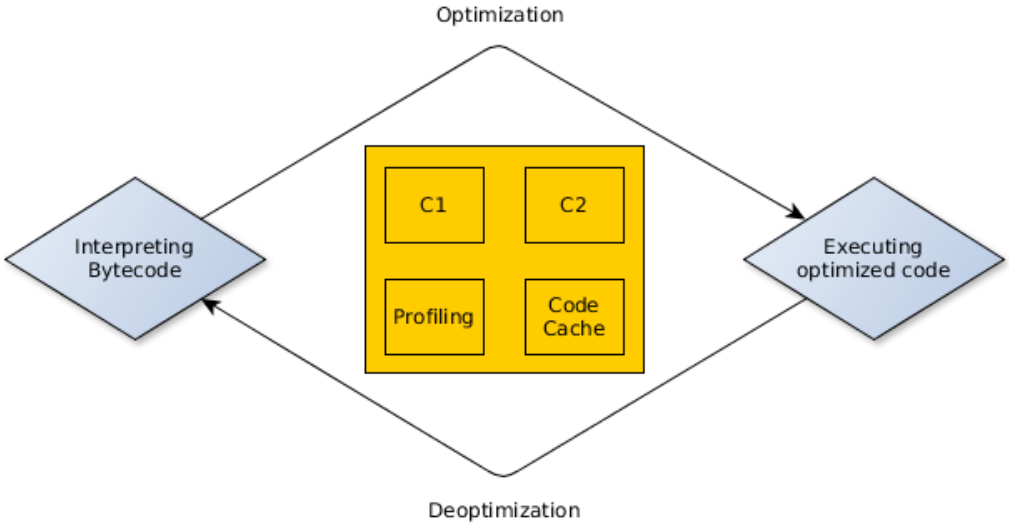

Рисунок 15 Взаимодействие двух компиляторов внутри JVM

JVM использует интерпретатор для выполнения кода (без оптимизации) сразу после запуска и профилирует его, чтобы решить, какие оптимизации применять. Он отслеживает количество вызовов для каждого метода, и если он превышает порог компилятора c1, он делает способ, подходящий для оптимизации компилятором c1, поэтому он ставит в очередь метод компиляции. Аналогично для c2, если количество вызовов достигает определенного порога, в конечном итоге оно будет скомпилировано им.

1.16Оптимизация кода

Оптимизации можно применять и для обратных ситуаций. VM подсчитывает, сколько раз цикл завершает выполнение, и аналогично для количества вызовов метода он ставит в очередь тело цикла для компиляции после нескольких исполнений. Этот метод называется On Stack Replacement (OST), и он позволяет виртуальной машине переключаться между различными реализациями (интерпретируемыми или оптимизированными) работающего кода.

Многие оптимизации основаны на образованных предположениях, сформированных ранее собранными данными времени выполнения. Позже,

если появится новая информация, которая доказывает, что такое предположение ошибочно, VM деоптимизирует связанный код, чтобы предотвратить неправильное поведение. JVM проверяет предположения, сделанные для оптимизации с необычными ловушками. Это заставляет виртуальную машину возвращаться к интерпретируемому режиму и перекомпилировать затронутый код без неправильного предположения.

Скомпилированный оптимизированный код хранится в специальной куче, называемой кэшем кода. Он хранится там до тех пор, пока соответствующий класс не будет выгружен или пока код не будет отменен. Кэш-код кода имеет конечный размер, поэтому, если в нем заканчивается свободное пространство, дальнейший код не может быть скомпилирован, что может вызвать проблемы с производительностью. По моему опыту это редко случается даже для больших приложений, но если значение по умолчанию оказывается слишком маленьким для вашего приложения или вы намеренно сокращаете размер кеша кода, JVM начинает жаловаться на него:

Итак, подведем итоги, JVM откладывает компиляцию до того момента, когда станет доступно больше информации о целевой архитектуре и динамическом поведении кода. Данные пункты отличают динамическую компиляцию Just-In-Time от статической компиляции Ahead-Of-Time.

- Возможность оптимизации, учитывающей информацию и статистику времени выполнения
- Учет факта, что приложение имеет более медленное время запуска, и достигает своей максимальной производительности позже
- Факт того, что вам не нужно самостоятельно компилировать свой код на разные платформы, это будет сделано виртуальной машиной.

JVM - это блестящая часть программного обеспечения, которая заставляет ваш код работать быстрее, не беспокоясь о оптимизации, которая

может возникнуть за счет поддерживаемой кодовой базы (например, встраивания методов в исходный код – Inline методы). На эти оптимизации влияют многие факторы, поэтому не стоит ничего оптимизировать, прежде чем у вас не появится в этом необходимость. Чувства кишки о производительности кода могут вводить в заблуждение, потому что под капотом JVM намного больше, чем кажется. И в этом самое главное: в большинстве случаев многие причины высокой или низкой производительности не видны с более высоких уровней абстракции, поэтому вы можете сконцентрироваться на написании простых, элегантных и поддерживаемых приложений на Java, Scala, Kotlin или любом языке, который работает на JVM и не заботится о низкоуровневой оптимизации.

1.17Сборщик мусора

Управление памятью - это процесс распознавания, когда выделенные объекты больше не нужны, освобождая память, используемую такими объектами, и делает ее доступной для последующих распределений. В некоторых языках программирования, управление памятью - ответственность программиста. Сложность такой задачи приводит ко многим распространенным ошибкам, которые могут вызвать неожиданное или ошибочное поведение программы и сбои. Как результат, большая часть времени разработки часто проводится отладки и попытки исправить такие ошибки.

Одна из проблем, которая часто возникает в программах с явным управлением памятью, - это болтающиеся ссылки. Это может привести к тому, что будет освобождено пространство, используемое объектом, к которому имеет ссылку другой объект. Если объект с этой (болтающейся) ссылкой пытается получить доступ к исходному объекту, но пространство было

перераспределено на новый объект, результат будет непредсказуем и поведение программы может быть неожиданным.

Другой распространенной проблемой с явным управлением памятью является утечка пространства. Эти утечки возникают, когда память выделена, но ссылки уже не используются и новое пространство не может быть выделено. Например, если вы намерены освободить пространство, используемое для связанного списка, но вы делаете ошибку, просто освобождая первый элемент списка, остальные элементы списка больше не ссылаются, но они выходят за пределы доступности в программе и не могут использоваться или высвободится самостоятельно. Если утечек достаточно, они могут продолжать потреблять память до тех пор, пока вся доступная память не будет исчерпана.

Альтернативный подход к управлению памятью, который в настоящее время широко используется, особенно большинством современных объектно-ориентированных языков, является автоматическое управление памятью программы, называемое сборщиком мусора. Автоматическое управление памятью позволяет увеличить абстрагирование интерфейсов и придает коду большую надежность.

Сбор мусора избегает проблемы с оборванными ссылками, поскольку объект, который все еще упоминается где-то никогда не будет собираться мусором и поэтому не будет считаться свободным. Сбор мусора также решает проблему утечки пространства, описанную выше, поскольку она автоматически освобождает всю память, на которую больше не ссылаются.

1.17.1 Концепции сборки мусора

Основными задачами сборщика мусора являются:

- распределение памяти
- обеспечение того, чтобы все объекты, на которые ссылаются, остались в памяти

- восстановление памяти, используемой объектами, которые больше не доступны из ссылок при выполнении кода

Объекты, на которые ссылаются, считаются живыми. Объекты, на которые больше не ссылаются, считаются мертвыми и являются так называемым мусором. Процесс поиска и освобождения (также известного как регенерация) пространства, используемого этими объектами известен как сбор мусора.

Сбор мусора решает многие, но не все проблемы с распределением памяти. Например, вы можете создавать объекты на неопределенный срок и продолжать ссылаться на них, пока не будет доступно больше памяти. Сбор мусора также сложная задача, требующая времени и ресурсов.

Точный алгоритм, используемый для организации памяти, выделения и освобождения пространства, обрабатывается сборщиком мусора и скрыт от программиста. Пространство обычно выделяется из большого пула памяти, такого как куча.

Время сбора мусора зависит от сборщика мусора. Как правило, вся куча или ее часть очищаются после того, как пространство стало заполненным, либо при достижении порогового процента заполненности.

Задача выполнения запроса на распределение, которая включает в себя поиск блока неиспользуемой памяти определенного размера в куче, очень сложная. Основная проблема большинства динамических алгоритмов распределения памяти заключается в том, чтобы избежать фрагментации (см. ниже), сохраняя при этом возможности как распределения, так и освобождения.

1.17.2 Характеристики

Сборщик мусора должен быть как безопасным, так и всеобъемлющим. То есть, живые данные никогда не должны быть ошибочно освобождены и

мусор не должен оставаться невостребованным для более чем небольшого числа циклов сбора.

Также желательно, чтобы сборщик мусора работал эффективно, без введения длинных пауз, в течение которых приложение не работает. Однако, как и в большинстве компьютерных систем, часто возникают компромиссы между времени, пространства и частоты. Например, если размер кучи мал, сбор будет быстрым, но куча будет заполнена быстрее, что требует более частых коллекций. И наоборот, большая куча займет больше времени, чтобы заполнить и поэтому коллекции будут менее частыми, но они могут занять больше времени.

Другой желательной характеристикой сборщика мусора является ограничение фрагментации. Когда память для объекта освобождаются, свободное пространство может появляться в небольших кусках в различных областях, однако данного пространства может быть недостаточно для размещения большого объекта. Один подход к устранению фрагментации называется уплотнением. Данный подход имеет различные реализации, в зависимости от дизайна сборщика мусора.

Масштабируемость также важна. Распределение не должно стать узким местом масштабируемости для многопоточных приложений на многопроцессорных системах и сбор также не должны быть узким местом.

1.17.3 Варианты дизайна

Существует несколько возможных вариантов дизайнов сборщика мусора.

1.17.3.1 Последовательный против параллельного

С последовательной сборкой мусора используется только один ЦП. Например, даже когда несколько ЦП доступно, только один используется для выполнения сборки мусора. Когда используется параллельный сбор, задача сбора мусора разделяется на части, и эти части выполняются одновременно,

на разных ЦП. Одновременная операция на нескольких ЦП позволяет сделать сбор быстрее, за счет некоторой дополнительной сложности и потенциальной фрагментации.

1.17.3.2 Параллельный против пауз в приложении

Когда выполняется сбор с паузами, выполнение приложения полностью приостанавливается во время сбора. Как альтернатива, одна или несколько задач сбора мусора могут выполняться параллельно, то есть одновременно с приложением. Как правило, параллельный сборщик мусора выполняет большую часть своей работы параллельно, но иногда также может потребоваться несколько коротких пауз в работе приложения. Сбор мусора с паузами в приложении проще, чем параллельная сборка, поскольку куча заморожена, и объекты не меняются во время сбора. Недостатком является то, что для некоторых приложений может быть нежелательно приостановить работу. Соответственно, время паузы короче, когда сбор мусора выполняется параллельно, но сборщик должен проявлять особую осторожность, поскольку он работает над объектами, которые могут быть параллельно обновлены приложением. Это добавляет некоторые накладные расходы для параллельных сбопрщиков мусора, которые влияют на производительность и требуют большего размера кучи.

1.17.3.3 Компактность по сравнению с некомпактным и копированием

После того, как сборщик мусора определил, какие объекты в памяти живые, а какие - мусор, он может сжимать память, перемещать все живые объекты вместе и полностью освобождать оставшуюся память. После уплотнения легко и быстро выделить новый объект в первом свободном месте. Простой указатель можно использовать для отслеживания следующего местоположения, доступного для размещения объекта. В отличие от уплотняющего коллектора, некомпактный коллектор освобождает пространство, используемое мусорными объектами на месте, т. Е. Не

перемещает все живые объекты, чтобы создать большую область регенерации таким же образом, как это делает сборщик уплотнений. Преимуществом является ускоренное завершение сбора мусора, но недостатком является потенциальная фрагментация. В общем, дороже выделять из кучи с освобождением места на месте, чем из уплотненной кучи. Может потребоваться выполнить поиск в куче для смежной области памяти, достаточно большой для размещения нового объекта. Третьей альтернативой является копировальный коллектор, который копирует (или эвакуирует) живые объекты в другую область памяти. Преимущество состоит в том, что исходная область может считаться пустой и доступной для быстрого и легкого последующего распределения, но недостатком является дополнительное время, необходимое для копирования, и дополнительное пространство, которое может потребоваться.

1.17.4 Производительность

Для интерактивного приложения может потребоваться маленькое время паузы, тогда как общее время выполнения более важно для не интерактивного. Приложение в режиме реального времени потребует небольших верхних границ, как для пауз сбора мусора, так и для доли времени, потраченного в сборщике в любой период. Малая занимаемая площадь может быть главной проблемой приложения, работающего на небольшом персональном компьютере или встраиваемой системе.

1.17.5 Разделение сборки на поколения

Метод, когда применяется так называемая сборка поколений, память делится на поколения, то есть отдельные пулы, в которых хранятся объекты разного возраста. Например, наиболее широко используемая конфигурация имеет два поколения: один для юных объектов и один для старых объектов.

Различные алгоритмы могут использоваться для сбора мусора в разных поколениях, каждый алгоритм оптимизирован на основе общепринятых

характеристик для данного поколения. Коллективная сборка мусора использует следующие наблюдения, известные как гипотеза слабой генерации, относительно приложений, написанных на нескольких языках программирования, включая язык программирования Java:

- Большинство выделенных объектов не ссылаются (считаются живыми) надолго, то есть они умирают молодыми
- Существует несколько ссылок от более старых к более молодым объектам

Сборка молодого поколения происходят относительно часто и являются эффективными и быстрыми, потому что пространство молодого поколения обычно невелико и, вероятно, содержит много объектов, на которые больше не ссылаются.

Объекты, которые выживают в нескольких сборках молодого поколения, в конечном итоге продвигаются до старого поколения. Это поколение обычно больше, чем молодое поколение, и его заполнение растет медленнее. В результате сборки старых поколений происходят нечасто, но для этого требуется значительно больше времени.

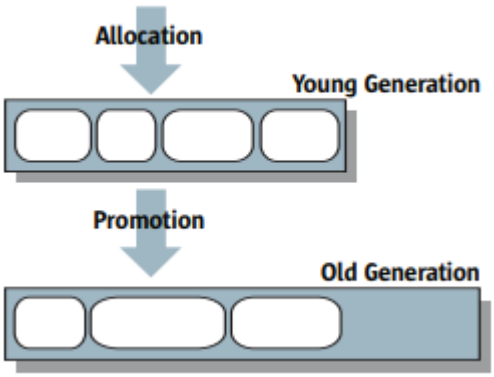

Рисунок 16 Смена поколения объектов в памяти

Алгоритм сбора мусора, выбранный для молодого поколения, как правило, повышает скорость, поскольку сборка молодого поколения часты. С другой стороны, старое поколение, как правило, управляется алгоритмом, который более эффективен в пространстве, потому что старое поколение

занимает большую часть кучи, а алгоритмы старого поколения должны хорошо работать с низкой плотностью мусора.

3. Сбор данных

Для проверки возможности применения математической модели для планирования мощностей для интеграционного решения, необходимо разобрать варианты получения данных на имеющихся типовых приложениях.

1.18Тестирование ПО

Тестирование программного обеспечения (Software Testing) - проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом [14]. В более широком смысле, тестирование - это одна из техник контроля качества, включающая в себя активности по планированию работ (Test Management), проектированию тестов (Test Design), выполнению тестирования (Test Execution) и анализу полученных результатов (Test Analysis).

Рассмотрим подробнее два вида тестирования ПО: статическое и динамическое.

Статическое тестирование (static testing): Тестирование, при котором элемент тестирования анализируется с использованием совокупности критериев качества или других свойств без выполнения кода. Пример - Ревизия, статический анализ [13].

Динамическое тестирование (dynamic testing): Тестирование, при котором требуется выполнение элемента тестирования [13]. Динамическое тестирование включает в себя следующие процессы:

Рисунок 17 Схема процессов динамического тестирования

Процессы динамического тестирования можно рассматривать на различных уровнях [13] (Рисунок 17):

- уровень покомпонентного тестирования
- уровень интеграционного теста
- уровень тестирования системы
- уровень приемочного испытания.

Компонентный (модульный) уровень тестирования позволяет проверить функционирование отдельно взятого элемента системы. Что считать элементом – модулем системы определяется контекстом. Наиболее полно данный вид тестов описан в стандарте IEEE 1008-87 “Standard for Software Unit Testing”, задающем интегрированную концепцию систематического и документированного подхода к модульному тестированию [15].

Интеграционный уровень тестирования является процессом проверки взаимодействия между программными компонентами/модулями. Классические стратегии интеграционного тестирования – “сверху-вниз” и “снизу-вверх” – используются для традиционных, иерархически структурированных систем и их сложно применять, например, к тестированию слабосвязанных систем, построенных в сервисно-ориентированных архитектурах (SOA) [15]. Современные стратегии в большей степени зависят

от архитектуры тестируемой системы и строятся на основе идентификации функциональных “потоков” (например, потоков операций и данных). Интеграционное тестирование – постоянно проводимая деятельность, предполагающая работу на достаточно высоком уровне абстракции. Наиболее успешная практика интеграционного тестирования базируется на инкрементальном подходе, позволяющем избежать проблем проведения разовых тестов, связанных с тестированием результатов очередного длительного этапа работ, когда количество выявленных дефектов приводит к серьезной переработке кода.

Системное тестирование охватывает целиком всю систему. Большинство функциональных сбоев должно быть идентифицировано еще на уровне модульных и интеграционных тестов [15]. В свою очередь, системное тестирование, обычно фокусируется на нефункциональных требованиях – безопасности, производительности, точности, надежности т.п. На этом уровне также тестируются интерфейсы к внешним приложениям, аппаратному обеспечению, операционной среде и т.д.

Приемочное тестирование проверяет поведение системы на предмет удовлетворения требований заказчика. Это возможно в том случае, если заказчик берет на себя ответственность, связанную с проведением таких работ, как сторона “принимающая” программную систему, или специфицированы типовые задачи, успешная проверка (тестирование) которых позволяет говорить об удовлетворении требований заказчика. Такие тесты могут проводиться как с привлечением разработчиков системы, так и без них [15].

Остановимся подробнее на системном уровне тестирования, так как на данном уровне рассматриваются требования к производительности. Для тестирования производительности выделяют следующие типы тестирования [13]:

- Тестирование потенциальных возможностей (capacity testing)
- Тестирование износостойкости (endurance testing)

- Нагрузочное тестирование (load testing)
- Стрессовое тестирование (stress testing)
- Объемное тестирование (volume testing)

Тестирование потенциальных возможностей (capacity testing): Тип тестирования уровня производительности для оценки уровня, при котором с увеличением нагрузки (числа пользователей, транзакций, элементов данных и т.д.) элемент тестирования подвергается угрозе не обеспечивать требуемую производительность [13].

Тестирование износостойкости (endurance testing): Тип тестирования уровня производительности для определения того, может ли элемент тестирования постоянно выдерживать требуемую нагрузку в течение установленного периода времени [13].

Нагрузочное тестирование (load testing): Тип тестирования уровня производительности, проводимого для оценки поведения элемента тестирования при ожидаемых условиях переменной нагрузки, обычно для ожидаемых условий низкого, типичного и пикового использования [13].

Стрессовое тестирование (stress testing): Тип тестирования уровня производительности, проводимого для оценки поведения элемента тестирования при условиях загрузки, выше ожидаемой или указанной в требованиях к производительности, или при доступности ресурсов, ниже минимальной, указанной в требованиях [13].

Объемное тестирование (volume testing): Тип тестирования уровня производительности, проводимого для оценки способности элемента тестирования обработать определенные объемы данных (обычно равных или близких к максимальным указанным потенциальным возможностям) с точки зрения потенциальных возможностей пропускной способности, емкости памяти или того и другого [13].

1.19Обзор ПО для тестирования

Исходя из определения типов тестирования для замеров показателей потребления ресурсов следует использовать инструменты, специализирующиеся на нагрузочном тестировании.

Специализированных систем нагрузочного тестирования для сервисных шин предприятия нет, но так как сервисная шина предприятия предоставляет доступ к функционалу посредствам web-сервисов, то для тестирования сервисной шины предприятия достаточно использовать инструмент нагрузочного тестирования для web-сервисов:

Наименование	Плюсы	Минусы
Apache HTTP server benchmarking tool	Не требует никакой дополнительной настройки; Очень простой инструмент.	Очень простой инструмент; Тестирует только производительность веб-сервера: опрашивает только один URL, не поддерживает сценарии нагрузки, невозможно имитировать пользовательскую нагрузку и оценить работоспособность проекта со всех сторон - как с точки зрения инфраструктуры, так и с точки зрения разработки.

Joe Dog Siege	Многопоточный; Можно задавать как количество запросов, так и продолжительность (время) тестирования - т.е можно эмулировать пользовательскую нагрузку; Поддерживает простейшие сценарии	Ресурсоемкий; Мало статистических данных и не очень хорошо эмулирует такие пользовательские сценарии, как ограничение скорости запросов пользователя; Не подходит для серьезного и масштабного тестирования в сотни и тысячи потоков, т.к он сам по себе ресурсоемкий, а на большом количестве запросов и потоков очень сильно нагружает систему.
Apache JMeter	Кроссплатформенный, т.к написан на Java; Очень гибкий, используется много протоколов, не только веб-сервер, но и базы; Управляется через консоль и gui интерфейс; Использование напрямую логов веб-сервера Apache и Nginx в качестве сценария с возможностью	Ресурсоемкий; На длительных и тяжелых тестах часто падает по разным причинам; Стабильная работа зависит от окружения и конфигурации сервера.

	варьирования нагрузки по этим профилям; Достаточно удобный и мощный инструмент.	
WAPT	Очень гибкий, большое количество настроек и тестов; Эмуляция медленных каналов соединений пользователей; Подключение модулей; Запись сценариев тестирования прямо из браузера, как с десктопного, так и с мобильного; Генерация различных графиков тестирования с помощью скриптов.	Доступен только для Windows; Платный.

Для оценки мониторинга потребления оперативной памяти и загрузки центрального процессора, инструмент нагрузочного тестирования также

должен иметь возможность замерять данные показатели на той физической машине, где установлено интеграционное решение. Следует отметить, что клиентские запросы должны отправляться с других физических машин. Не один из перечисленных инструментов не предоставляет такой возможности. Исходя из этого факта было разработана тестирующая система.

1.20Тестирующая система

Основной задачей ESB является предоставление единого унифицированного доступа к данным предприятия посредством web-сервисов для множества клиентских приложений, имеющих доступ к данному сервису [3]. Этот факт очень важен при проектировании тестирующего приложения, так как накладывает ограничение по минимальному количеству приложений, одновременно запрашивающих данные посредствам web-сервисов. Сервисная шина предприятия должна корректно обрабатывать множественные запросы к web-сервису, а также эффективно тратить системные ресурсы на обслуживание данных запросов. Под ресурсами стоит понимать оперативную память и загрузку центрального процессора. Замеры ресурсов должны производиться на той же физической машине, где установлена и запущена сервисная шина предприятия. Данные замеры стоит производить периодически с определенным временным промежутком в фоновом режиме. Замеры затрачиваемых ресурсов должны минимально сказываться на производительности системы.

Тот факт, что сервисная шина предприятия в реальных ситуациях обслуживает несколько клиентских приложений в многопоточном режиме, заставляет разбить тестирующие приложение на управляющие Master приложение и тестирующие Slave приложение [17].

Основной задачей Master приложения является контроль всех Slave приложений, а также контроль за выполнением хода замеров ресурсов, используемых сервисной шиной предприятия.

Основной задачей Slave приложения является отправка http запросов с заданными параметрами. Отправку запросов приложение осуществляет периодически, что позволяет имитировать непрерывное использование web-сервисов сервисной шины предприятия. Предполагается, что в одном тестовом окружении будет запущено несколько копий клиентского приложения, каждая из которых должна работать на отдельной физической машине.

Исходя из факта, что процесс тестирования сервисной шины предприятия управляется одним приложением, расположенном на другой физической машине, было принято решение о дополнительных требованиях к сервисной шине предприятия:

- Замеры должны производиться отдельным программным модулем
- Запуск процесса периодических замеров ресурсов должен осуществляться удаленно
- Остановка процесса периодических замеров ресурсов должен осуществляться удаленно
- Получение результатов процесса периодических замеров ресурсов должен осуществляться удаленно

Архитектура системы представляет собой классическую SOA архитектуру [18], где выделяются следующие сервисы:

- ESB
- Master приложение
- Slave приложения

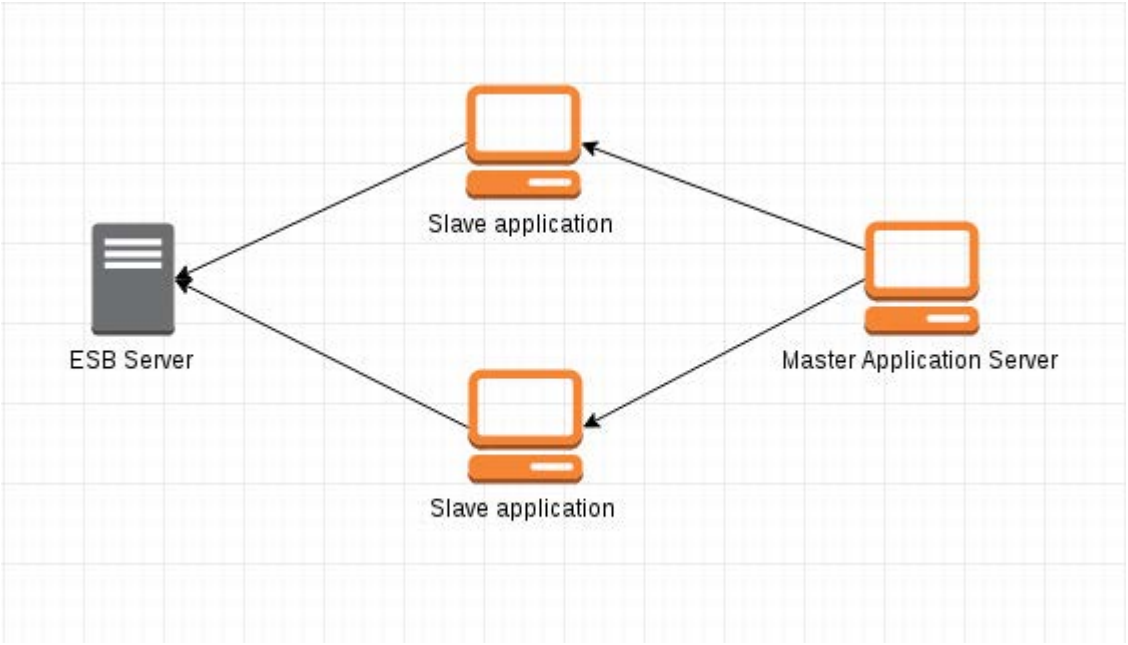

Рисунок 18 Архитектура тестирующей системы

Для обеспечения непрерывного замера показателей в определенном временном промежутке с минимальными влияние на использование ресурсов целевой системы была разработана библиотека Resource Usage Monitor. Данная библиотека позволяет делать замеры потребления ресурсов системы, изменять промежуток между замерами, а также изменять размер буфера данных. Буфер данных необходим для обеспечения минимального влияния на использование ресурсов во время замера.

Библиотека является набором Java классов и поэтому может быть использована напрямую из сервисной шины данных, что уменьшит затраты на замер ресурсов сервисной шины предприятия.

Библиотека может быть использована в любом JVM совместимом языке программирования. Библиотека может быть использована как Apache Maven зависимость [19], а также имеет ряд дополнительных Java классов, используемых во многих приложениях (Листинг 1).

Пример использования:

```java
//create instance with 10ms period and 10 000 buffer length
ResourcesUsageMonitor resourcesUsageMonitor = new ResourcesUsageMonitor(10, 10000);
resourcesUsageMonitor.initialize();

//start monitoring
resourcesUsageMonitor.start();

//sleep 10s
Thread.sleep(10000);

//stop monitoring
resourcesUsageMonitor.stop();
//get results
List<State> result = resourcesUsageMonitor.getResult();

//print some properties of results set
State max = result.stream().max(Comparator.comparingLong(x -> x.getUsedMemory())).get();
System.out.println(max);
```

Листинг 1 Использование библиотеки для замеров использования ресурсов

Так как большинство операций распределенной системы осуществляется посредством вызова web-сервисов, то следует заранее регламентировать протокол общения между ключевыми элементами системы:

- Клиентское приложение
- Сервисная шина предприятия
- Master приложение
- Slave приложение
- Клиентское приложение должно иметь следующие возможности:
- Регистрация сервисной шины предприятия для тестирования
- Регистрация Slave приложение в Master приложении
- Получение списка доступных тестов в Master приложении
- Запуск процесса тестирования
- Получение результатов тестирования
- Master приложение должно иметь следующие возможности:
- Запрос списка поддерживаемых тестах в сервисной шине предприятия
- Запрос статуса Slave приложения

- Запуск процесса периодических замеров в сервисной шине предприятия
- Запуск отправки http запросов в Slave приложении
- Остановка процесса периодических замеров в сервисной шине предприятия
- Slave приложение должно иметь следующие возможности:
- Отправка произвольного http запроса
- Отправка результатов тестирования в Master приложение

Общение между частями системы осуществляется с помощью http запросов методом POST. Параметры вызова передаются в теле запроса в виде JSON объекта [20].

Примеры запросов-ответов для клиентского приложения, в случае если тело запроса не указано - используется метод GET.

Ответ	Тело запроса	Описание

{
"status": "OK"
}
	{
	"host": "localhost",
	"port": "8088"
	}
		Регистрация сервисной шины предприятия для тестирования

{
"status": "OK"
}
	{
	"host": "localhost",
	"port": "8090"
	}
		Регистрация Slave приложение в Master приложении

["inbound-soap", "outbound-soap", "outbound-rest", "inbound-rest", "outbound-ftp"]		Получение списка доступных тестов в Master приложении

{
"status": "OK"
}
	[
	{
	"name": "inbound-soap",
	"delay": 1000,
	"time": 2000
	}
	]
		Запуск процесса тестирования

{
"esbResults": {
"inbound-soap": [
{
"schedule": {
"workerType": "Http",
"endDate": "Jun 4, 2017 8:55:47 PM",
"start": "Jun 4, 2017 8:55:45 PM",
"name": "inbound-soap",
"additionalParameters": {...},
"id": "a70d0954-444b-4cb7-8be7-1b14f4f9c6a1"
},
"results": [
{
"usedMemory": 13451264,
"systemCpuLoad": 0.007488479262672811,
"stateNumber": 0
}
]
}
]
},
"clientResults": {
"inbound-soap": [
{
"schedule": {...},
"results": [
4
]
}
]
}
}
		Получение результатов тестирования

1.20.1 Master приложение

Основной задачей Master приложения является контроль всех Slave приложений, контроль за выполнением хода замеров ресурсов, используемых сервисной шиной предприятия, а также агрегация результатов тестирования.

Master приложение имеет следующие web-сервисы для взаимодействия с внешними узлами системы:

- post:/api/client/commit
- post:/api/client/register
- post:/api/esb/register
- get:/api/result
- get:/api/esb/testcases
- post:/api/esb/testcases/run

Master приложение разработано с помощью языка программирования Java 8 и использует следующий стек технологий для реализации web-сервисов:

- Apache Maven - сборка приложения, контроль зависимостей
- Spring framework - beans и beans injection [21]
- Google Gson - сериализация/десериализация [22]
- Внутренний пакет commons - использование общих DTO классов
- Spring Boot - обертка приложение в web-сервер Apache Tomcat [21]

Master приложение требует, чтобы сервисная шина данных предоставляла список реализованных тестов, дополнительной информацией, а именно:

- Имя теста
- Тип клиентского запроса (на данный момент поддерживается только http)
- Адрес запроса
- Http метод запроса

- Список http заголовков запроса
- Тело запроса, которое будут отправлять Slave приложения

При регистрации сервисной шины предприятия данная информация запрашивается и сохраняется во временной памяти приложения. Данная информация необходима для формирования:

- Списка доступных тестов
- Расписания запросов Slave приложения

При запуске тестирования формируется и отправляется Slave приложению запрос содержащий следующие данные:

- Время начала тестирования
- Тип клиентского запроса
- Время окончания
- Имя теста
- Адрес запроса
- Http метод запроса
- Список http заголовков запроса
- Тело запроса

Сформированное сообщение рассылается каждому из зарегистрированных Slave приложений.

1.20.2 Slave приложение

Основной задачей Slave приложения является отправка http запросов с заданными параметрами. Отправку запросов приложение осуществляет периодически, что позволяет имитировать непрерывное использование web-сервисов сервисной шины предприятия. Предполагается, что в одном тестовом окружении будет запущено несколько копий клиентского приложения, каждая из которых должна работать на отдельной физической машине.

Slave приложение имеет следующие web-сервисы для взаимодействия с внешними узлами системы:

- get:/api/healthcheck
- post:/api/schedule

Для отправки http запроса Slave приложение использует параметры, отправленные Master приложением, при создании расписания отправки запросов:

- Адрес запроса
- Http метод запроса
- Список http заголовков запроса
- Тело запроса

Системное время Slave приложения и Master приложения должны быть синхронизированы.

1.20.3 Пример использования

Распределенная система нагрузочного тестирования сервисных шин предприятия не имеет web интерфейса. Пользователю, для управления тестированием, доступны следующие web-сервисы, реализованные в Master приложении:

- post:/api/client/register
- post:/api/esb/register
- get:/api/result
- get:/api/esb/testcases
- post:/api/esb/testcases/run

Для работы с web-сервисами можно использовать любой http клиент с поддержкой POST и GET запросов, в данном примере используется Postman.

Рассмотрим пример тестирования сервисной шины предприятия Mule ESB с одним Slave приложением. Для удобства все приложения развернуты на одной физической машине.

Регистрация Slave приложения:

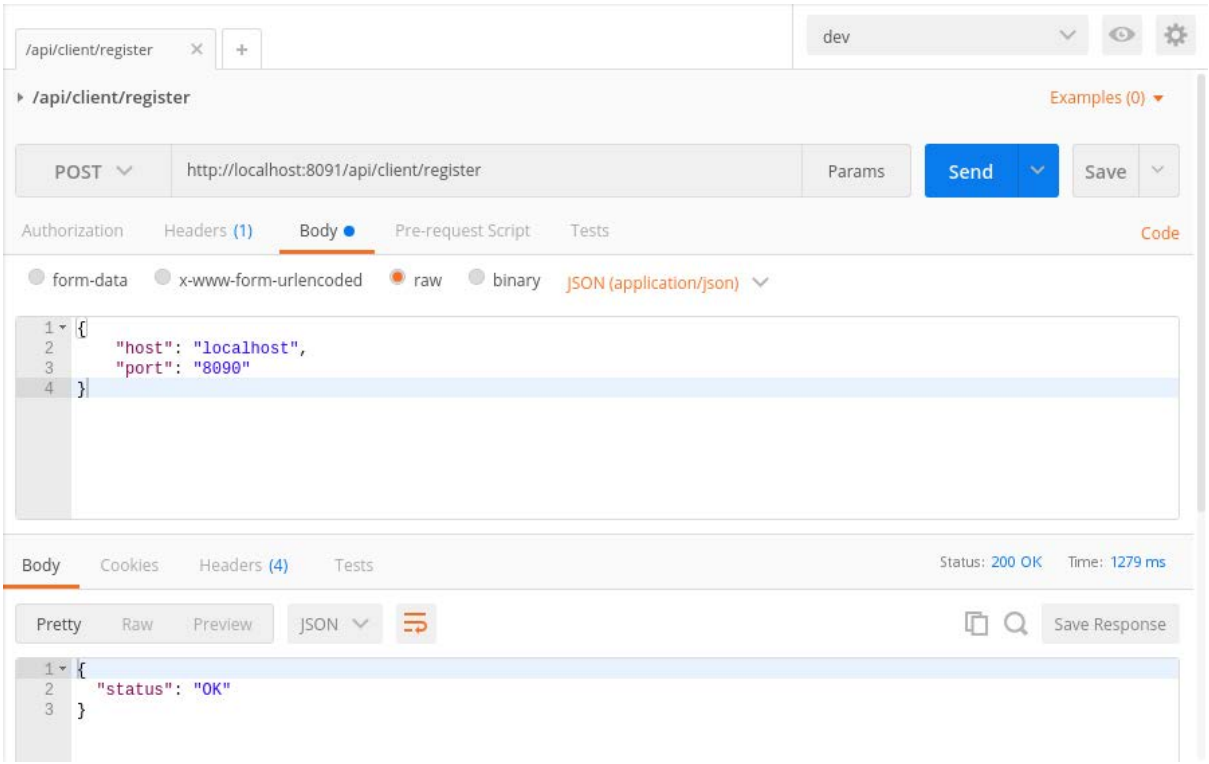

Рисунок 19 Регистрация Slave приложения

Регистрация сервисной шины предприятия:

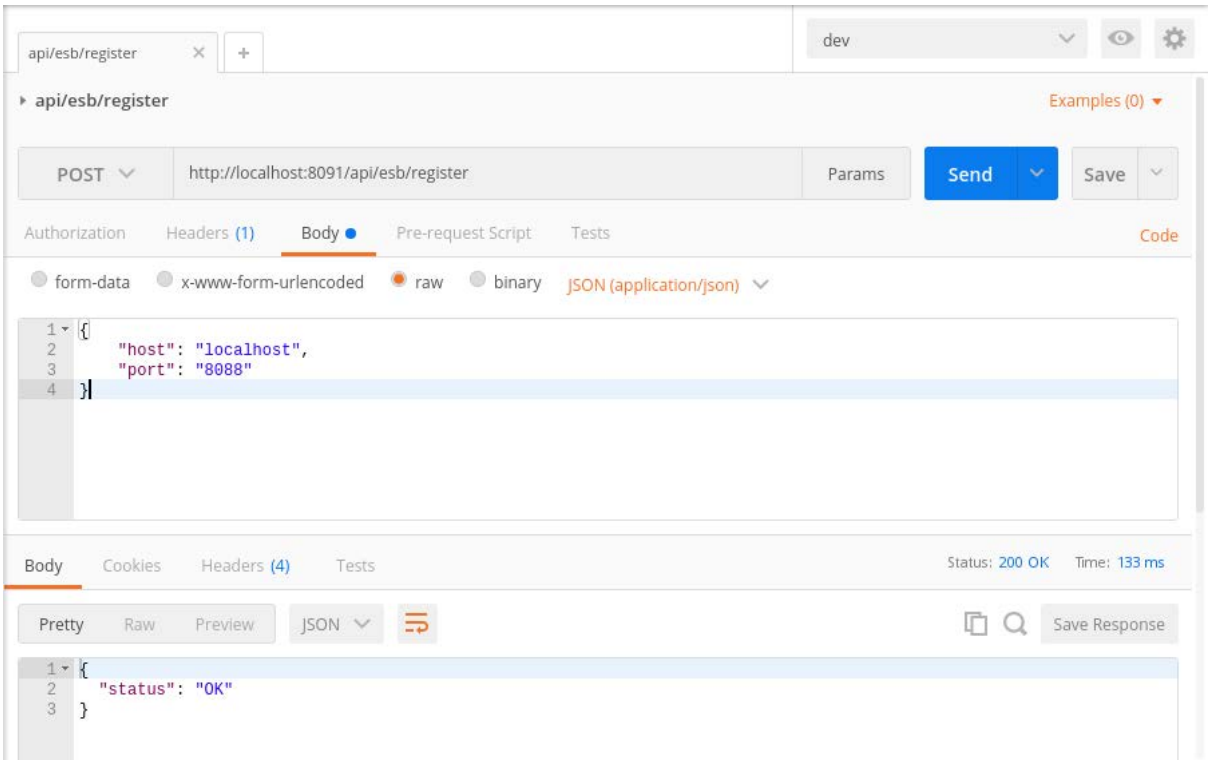

Рисунок 20 Регистрация сервисной шины предприятия

Просмотр доступных тестов:

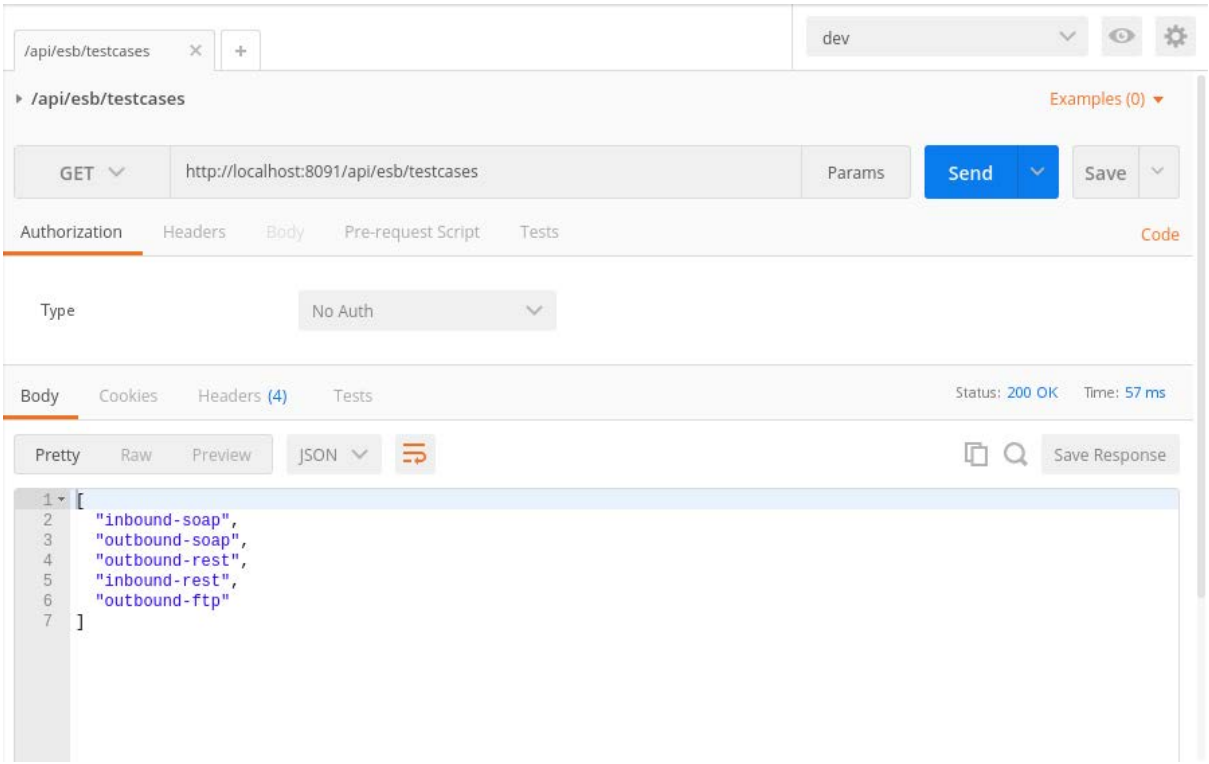

Рисунок 21 Просмотр доступных тестов

Просмотр результатов тестирования:

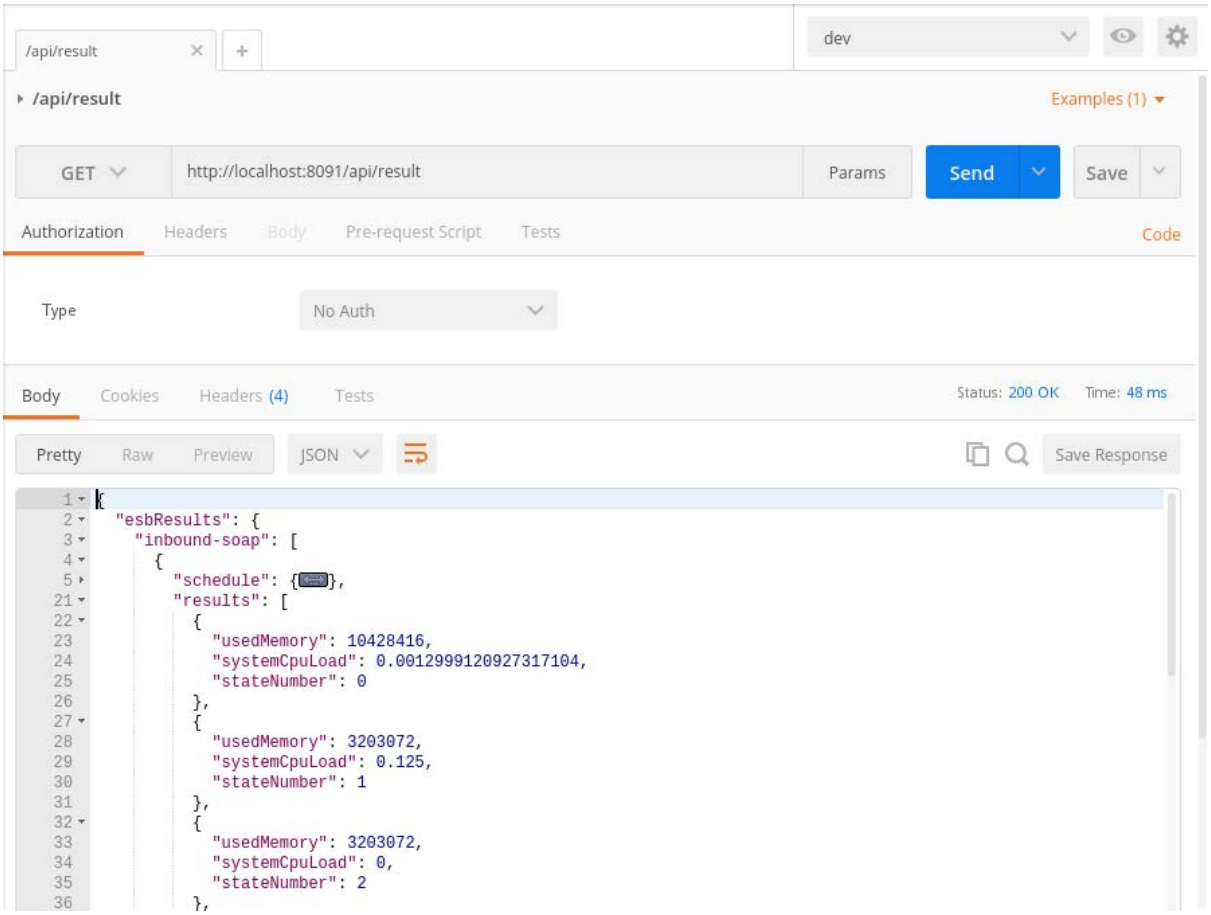

Рисунок 22 Просмотр результатов тестирования

4. Эксперимент

В данной статье рассматривается одна из задач интеграции систем, решаемых с помощью Сервисной Шины Предприятия: Uniform Data Access (Унифицированный доступ к данным) [3]

Для получения показателей потребления мощностей Сервисной Шиной Предприятия был выбран один из базовых сценариев ее использования, при задаче предоставления унифицированного доступа к данным: создание restful сервиса, где источником данных является база данных.

Была построена виртуальная лаборатория, включающая в себя следующие элементы

- База Данных (PostgreSQL)
- Интеграционное приложение (На базе Сервисной Шины Предприятия Apache Camel)
- Приложение для управление ходом эксперимента [4]
- Приложение для отправки запросов в интеграционное приложение [4]

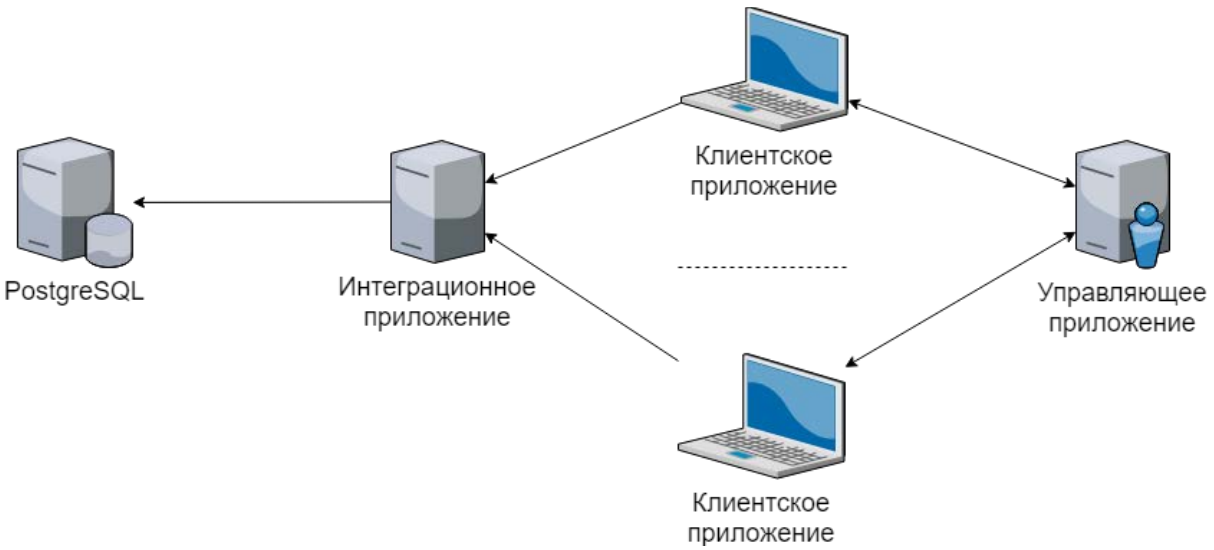

Рисунок 23 Схема компонентов виртуальной лаборатории

Основными шагами эксперимента являются:

- Установка и запуск клиентских приложений, управляющего, интеграционного и базы данных
- Подготовка данных для эксперимента. Для эксперимента были использованы данные агрегируемые из нескольких источников [5]
- Настройка Управляющего приложения
- Запуск эксперимента
- обработка и сохранения результатов в виде таблицы

Интеграционное приложение принимает на вход сообщения от клиента в виде json объекта [4]:

```
{
  "columnsCount": 5,
  "rowsCount": 10,
  "serializeType": "JSON"
}
```

Листинг 2 Запрос данных через интеграционное решение

На каждый такой запрос интеграционное приложение запускает фоновый процесс замера показателей ресурсов, создает динамический sql запрос и исполняет его. В зависимости от параметра “serialize Type” интеграционное приложение сериализует данные одним из трех способов:

- JSON
- XML
- CSV

Далее к полученному http телу ответа, добавляются http заголовки с результатами замеров показателей потребления ресурсов.

Клиентское приложение, при получении ответа от интеграционного приложения, отправляет результаты замеров в управляющее приложение. Таким образом все результаты замеров потребления ресурсов по каждому запросу консолидируются в управляющем приложении [4].

При планировании эксперимента были выделены следующие показатели потребления мощностей интеграционным приложением:

- Время отклика на каждый запрос

- Загрузка центрального процессора
- Использование оперативной памяти приложением
- Также были выделены следующие факторы влияющие на показатели потребления мощностей:
- serializeType - типа сериализации сообщения. Возможные значения: JSON, XML, CSV
- concurrentlyMonitoringCount - количество параллельных запросов, отправляемых клиентским приложением в интеграционное
- rowsCount - количество строк, выбранных из таблицы базы данных
- columnsCount - количество столбцов, выбранных из таблицы базы данных
- cpuCount – количество CPU в компьютере, на котором происходят замеры
- availableMemory – доступная операцитвая память компьютера, , на котором происходят замеры

По полученным факторам и откликам был составлен следующий план эксперимента:

Фактор	Основной уровень	Область определения	Интервал	Примеры значений через запятую
serializeType	-	JSON, XML, CSV	-	JSON, XML, CSV
concurrentlyMonitoringCount	6	[1, 10]	2	2,4,6,8,10
rowsCount	6000	[1, 11000]	1000	1000,2000,3000

columnsCount	30	[1,50]	10	10,20,30,40,50
cpuCount	4	[2;8]	2	2
availableMemory	8	[4;16]	4	8

1.21Результаты

Результаты эксперимента дополнительно были “очищены” от шумов.

Для выявления отсутствия зависимости откликов от факторов, была составлена таблица корреляции откликов от факторов:

	Время отклика	Загрузка цпу	Использования Оперативной памяти
serializeType	-0,32481	-0,15023	0,08665
concurrentlyMonitoringCount	0,61501	0,44862	0,70660
rowsCount	0,21164	0,07476	0,01089
columnsCount	0,48537	0,12886	0,00727
cpuCount	-0,435	-0,65	0,13
availableMemory	-0,11	-0,21	-0,23

Из данной таблицы видно, что все пары отклик/параметр запроса имеют зависимость.

Рассмотрим зависимость отклика от параметров запроса.

Рисунок 24 Среднее время отклика на запрос в зависимости от типа сериализации

Рисунок 25 Среднее время отклика на запрос в зависимости от количества параллельных запросов

Рисунок 26 Среднее время отклика на запрос в зависимости от количества элементов в ответе запроса

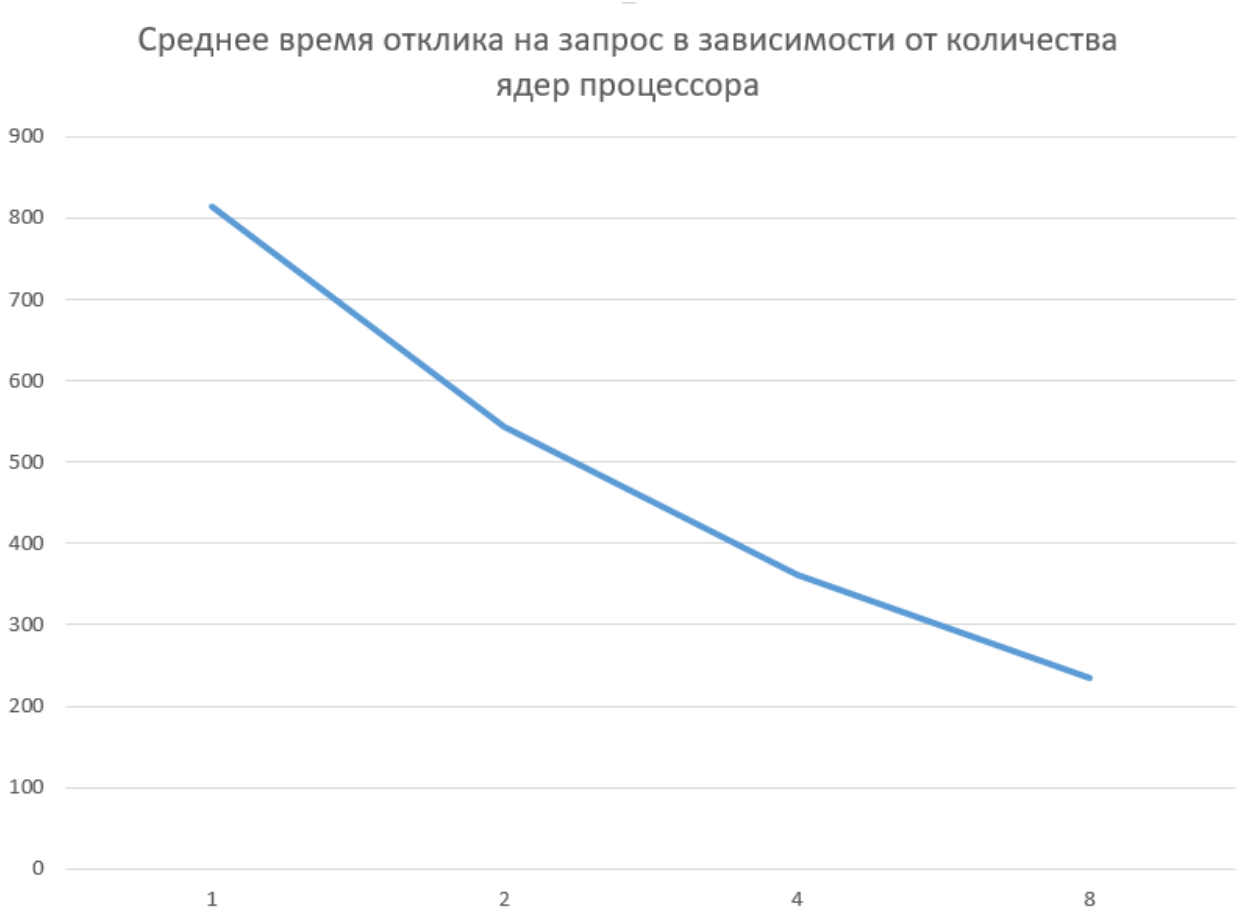

Рисунок 27 Среднее время отклика на запрос в зависимости от количества ядер процессора

Рисунок 28 Среднее время отклика на запрос в зависимости от объема оперативной памяти

1.22Прогнозирование мощностей

1.22.1Инструмент прогнозирования

Поскольку Python за последние годы приобрел большую популярность в отрасли Data Science, я хотел бы изложить некоторые из его наиболее полезных библиотек для ученых и инженеров по данным, основанных на недавнем опыте.

В данной работе были использованы следующие библиотеки:

1. SciKit-Learn (Коммиты: 21793, Авторы: 842). Scikits - это дополнительные пакеты SciPy Stack, предназначенные для определенных функций, таких как обработка изображений и упрощение машинного обучения. Что касается последнего, одним из наиболее значимых из этих пакетов является scikit-learn. Пакет построен на основе

SciPy и активно использует его математические операции. Scikit-learn предоставляет краткий и последовательный интерфейс для общих алгоритмов машинного обучения, что упрощает вывод ML в производственные системы. Библиотека сочетает качественный код и хорошую документацию, простоту использования и высокую производительность и является де-факто промышленным стандартом для машинного обучения с Python.

2. NumPy (Коммиты: 15980, Авторы: 522). Когда вы начинаете заниматься научной задачей в Python, к Вам может прийти на помощь Python SciPy Stack, который представляет собой набор программ, специально предназначенных для научных вычислений на Python (не путайте с библиотекой SciPy, которая является частью этого стека, и сообщество вокруг этого стека). Тем не менее, стек довольно обширен, в нем более десятка библиотек, и мы хотим поставить фокус на основные пакеты (особенно самые важные). Самый фундаментальный пакет, вокруг которого построен научный вычислительный стек, - NumPy (обозначает числовой Python). Он предоставляет множество полезных функций для операций над n-массивами и матрицами в Python. Библиотека обеспечивает векторизация математических операций по типу массива NumPy, что улучшает производительность и, соответственно, ускоряет выполнение.
3. SciPy (Коммиты: 17213, Авторы: 489). SciPy - это библиотека программного обеспечения для инженерии и науки. Не нужно путать SciPy Stack и SciPy Library. SciPy содержит модули для линейной алгебры, оптимизации, интеграции и статистики. Основная функциональность библиотеки SciPy построена на NumPy, поэтому ее массивы используют NumPy. Он обеспечивает эффективные численные подпрограммы, такие как численная интеграция, оптимизация и многие

другие через свои специфические подмодули. Функции во всех подмодулях SciPy хорошо документированы.

4. Pandas (Коммиты: 15089, Авторы: 762). Pandas - это пакет Python, предназначенный для простой и интуитивно понятной работы с «помеченными» и «реляционными» данными. Pandas - идеальный инструмент для работы с данными. Он предназначен для быстрой и простой манипуляции данными, агрегации и визуализации.

1.23 Модели прогнозирования

В ходе выбора модели для прогнозирования были опробованы следующие регрессионные модели:

- Generalized Linear Models
- Stochastic Gradient Descent – SGD
- Perceptron
- Support Vector Machines
- Gaussian Processes

1.23.1 Generalized Linear Models

Обобщённые линейные модели (GLM) - общепризнанный метод статистической обработки данных в страховании. Благодаря компьютерным технологиям, GLM широко признаны страховыми компании Европы, и быстро завоевывают признание профессионалов в США и Канаде.

GLM являются хорошо разработанным и простым для понимания способом построения моделей для анализа претензий и для прогнозирования продления старых/заключения новых страховых договоров.

1.23.2 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) — это метод, который подходит как для online обучения, или обучения в режиме реального времени, когда данные

поступают постепенно, так и для Big Data. Причина универсальности этого метода состоит в том, что SGD не нужен полный объем данных для решения задач классификации или регрессионного анализа: апдейт модели происходит постепенно, по мере того, как появятся новые данные.

Понять логику, каким образом это происходит можно на примере расчета средней величины потока данных:

- у нас есть текущее значение среднего μ
- если следующая величина X, которую мы получаем, больше чем μ, мы увеличиваем μ на $\tau*(X-\mu)$, где τ — это так называемый learning rate
- если следующая величина X меньше чем μ, то мы уменьшаем нашу оценку μ

1.23.3 Perceptron

Многослойными персептронами называют нейронные сети прямого распространения. Входной сигнал в таких сетях распространяется в прямом направлении, от слоя к слою. Многослойный персептрон в общем представлении состоит из следующих элементов:

- множества входных узлов, которые образуют входной слой;
- одного или нескольких скрытых слоев вычислительных нейронов;
- одного выходного слоя нейронов.

Многослойный персептрон представляет собой обобщение однослойного персептрона Розенблатта. Количество входных и выходных элементов в многослойном персептроне определяется условиями задачи. Сомнения могут возникнуть в отношении того, какие входные значения использовать, а какие нет. Вопрос о том, сколько использовать промежуточных слоев и элементов в них, пока совершенно неясен. В качестве начального приближения можно взять один промежуточный слой, а число

элементов в нем положить равным полусумме числа входных и выходных элементов.

1.23.4 Support Vector Machines

Машина опорных векторов — является одной из наиболее популярных методологий обучения по прецедентам, предложенной В. Н. Вапником и известной в англоязычной литературе под названием SVM (Support Vector Machine).

Оптимальная разделяющая гиперплоскость. Понятие зазора между классами (margin). Случай линейной разделимости. Задача квадратичного программирования. Опорные векторы. Случай отсутствия линейной разделимости. Функции ядра (kernel functions), спрямляющее пространство, теорема Мерсера. Способы построения ядер. Примеры ядер. Сопоставление SVM и нейронной RBF-сети. Обучение SVM методом активных ограничений. SVM-регрессия.

1.23.5 Gaussian Processes

В теории вероятностей и статистике гауссовский процесс - это стохастический процесс (совокупность случайных величин, индексированных некоторым параметром, чаще всего временем или координатами), такой что любой конечный набор этих случайных величин имеет многомерное нормальное распределение, то есть любая конечная линейная комбинация из них нормально распределена. Распределение гауссовского процесса – это совместное распределение всех его случайных величин и, в силу чего, является распределением функций с непрерывной областью определения.

Если рассматривать гауссовский процесс как способ решения задач машинного обучения, то используется ленивое обучение и мера подобия между точками (функция ядра) для получения прогноза значения невидимой точки из обучающей выборки. В понятие прогноза, помимо самой оценки

точки, входит информация о неопределенности — одномерное гауссовское распределение.

Для вычисления прогнозов некоторых функций ядра используют метод матричной алгебры, кригинг.

Гауссовский процесс назван так в честь Карла Фридриха Гаусса, поскольку в его основе лежит понятие гауссовского распределения (нормального распределения). Гауссовский процесс может рассматриваться как бесконечномерное обобщение многомерных нормальных распределений. Эти процессы применяются в статистическом моделировании; в частности, используются свойства нормальности. Например, если случайный процесс моделируется как гауссовский, то распределения различных производных величин, такие как среднее значение процесса в течение определенного промежутка времени и погрешность его оценки с использованием выборки значений, могут быть получены явно.

1.23.6 Выбор модели

Для выбора модели прогнозирования был использован следующий алгоритм:

- Обучение всех моделей на имеющихся данных
- Расчёт коэффициентов детерминации для каждой регрессионной модели
- Выбор модели на основе наибольшего отклика

В результате была получена следующая таблица:

Модель	Коэффициент детерминации
Generalized Linear Models	0.2
Stochastic Gradient Descent	0.535
Perceptron	0.815
Support Vector Machines	0.67
Gaussian Processes	0.1

Исходя из результатов, в качестве основной модели была выбрана модель: Perceptron.

1.23.7 Описание параметров модели

Для обучения регрессионной модели прогнозирования значения отклика от значений факторов были выбраны следующие параметры [6]:

- Функция активации линейная f(x) = x
- Квазиньютоновские методы для обучения многослойного перцептрона
- По данным полученным из эксперимента была обучена модель для получения значений:
  - Время отклика
  - Используемая память
  - Загрузка цп
  - На входе были переданы следующие параметры:
  - serializeType - типа сериализации сообщения. Возможные значения: JSON, XML, CSV
  - concurrentlyMonitoringCount - количество параллельных запросов, отправляемых клиентским приложением в интеграционное
  - rowsCount - количество строк, выбранных из таблицы базы данных
  - columnsCount - количество столбцов, выбранных из таблицы базы данных
  - cpuCount – количество CPU в компьютере, на котором происходят замеры
  - availableMemory – доступная операцитвая память компьютера, , на котором происходят замеры

1.24Результаты обучения

В процессе подбора оптимальных параметров модели прогнозирования были получены следующие результаты при моделировании отклика:

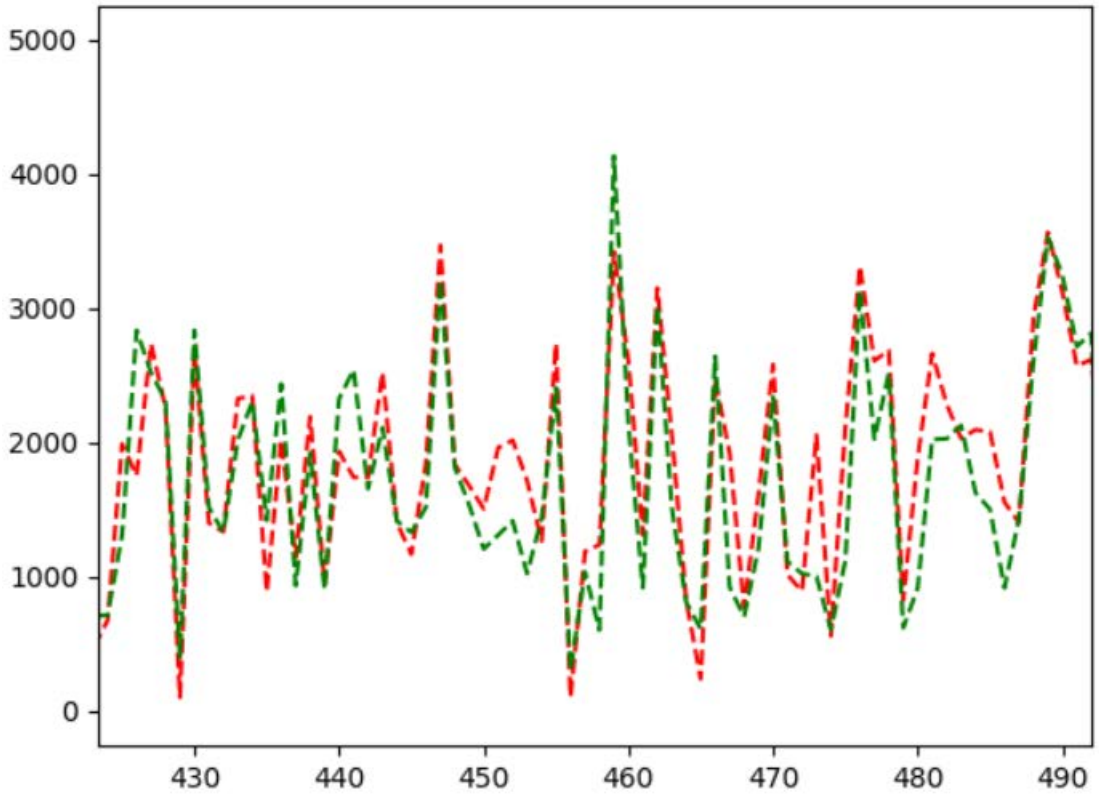

Рисунок 29 Линейная функция активации и квазиньютоновский алгоритм обучения

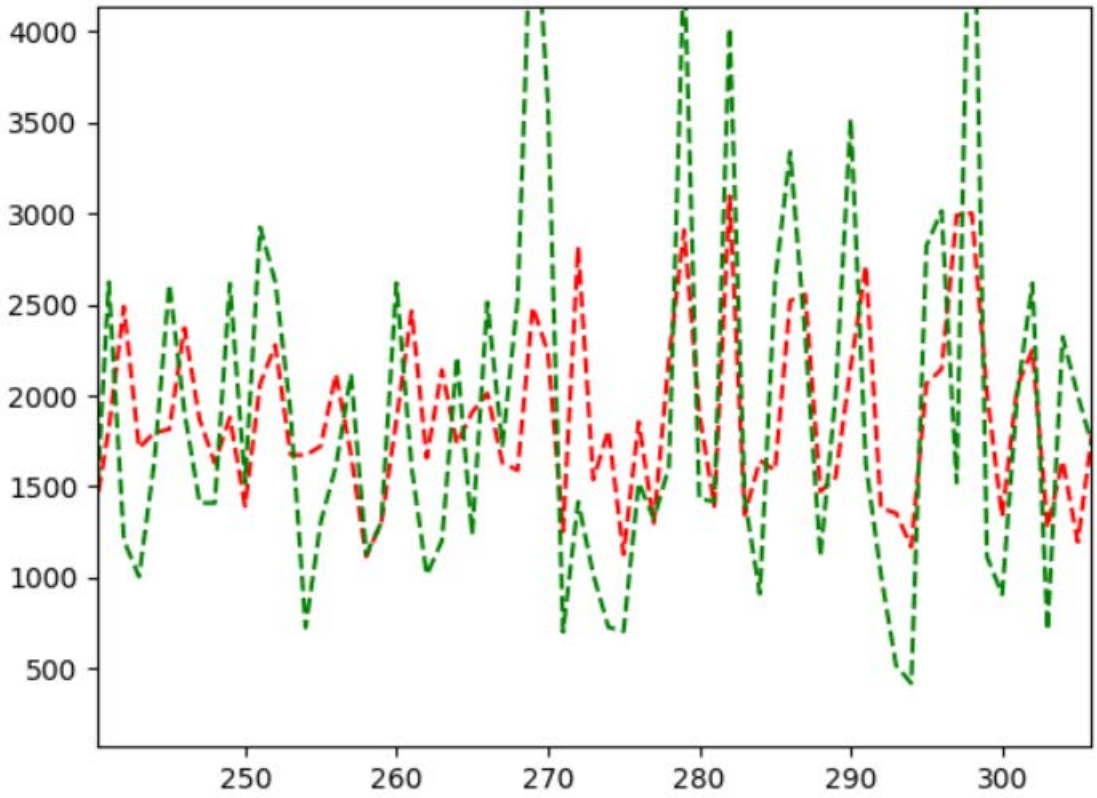

Рисунок 30 Линейная функция активации и метод Адама

Результаты обучения представлены в виде коэффициента детерминации для откликов, полученных с помощью обученной модели и откликов, полученных в ходе эксперимента:

Наименование выходного параметра	Значение коэффициента детерминации
Время отклика	0,835
Используемая память	0,674
Загрузка ЦП	0,26

Ниже представлен график “Время откликов на атомарный запрос” полученных при построении регрессионной модели (красный пунктир) и при эксперименте (зеленый пунктир):

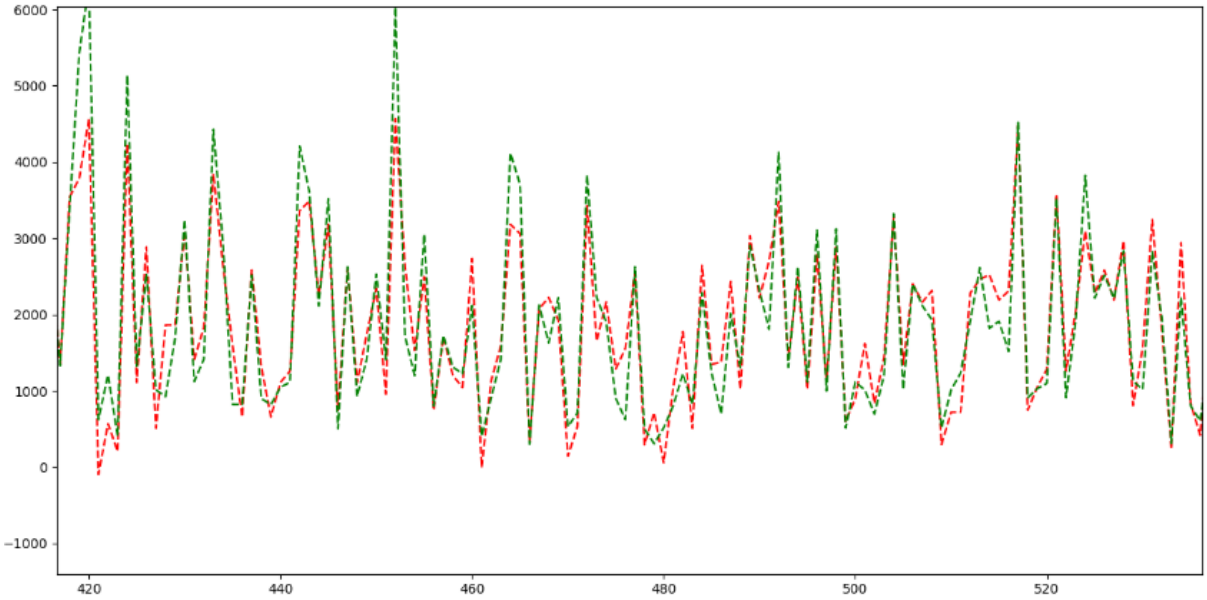

Рисунок 31 График отображающий результат обучения

Выводы

По результатам эксперимента и обучения модели можно сделать следующие выводы:

1. Данные полученные в виртуальной лаборатории пригодны для построения модели для прогнозирования, что позволяет решить задачу “Задача планирования производственных мощностей имеет” в этапе “Оптимизации плана компонентной конфигурации” [1]
2. Регрессионная модель способна достаточно хорошо прогнозировать время отклика на атомарный запрос
3. Регрессионная модель способна прогнозировать количество используемой памяти приложением. Но имеет достаточно большое отклонение: 51 мб. Требуется результаты эксперимента для большей загрузки, чтобы оценить, как поведет себя отклонение при больших загрузках на интеграционное приложение
4. Регрессионная модель для загрузки ЦП не может быть смоделирована, так как не учитывает следующие особенности работы приложения: работу “сборщика мусора”, JIT оптимизации приложений “на лету” и другие оптимизационные процессы, происходящие на уровне как Сервисной Шины Предприятия, так и на уровне виртуальной машины JAVA [23]

   Поставленная цель полностью достигнута.

Список литературы

1 Лаврищева Е.М. МЕТОДЫ И СРЕДСТВА ИНЖЕНЕРИИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ/Лаврищева Е.М. , Петрухин В.А. // Московский физико-технический институт (государственный университет), 2006. - С. 296

2 Луис Р.. Система канбан. Практические советы по разработке в условиях вашей компании / Пер. с англ. Е.В. Журиной; Под науч. ред. Э.А. Башкардина. — М.: РИА «Стандарты и качество» — 216 с., ил. — (Серия «Бережливое управление»),. 2008

3 Dittrich, K.R. Data Integration - Problems, Approaches, and Perspectives/Patrick Ziegler and Klaus R. Dittrich // Database Technology Research Group. Department of Informatics, University of Zurich. С. 4-5

4 Кабардинский Е.О. Выбор сервисной шины предприятия для разработки интеграционного решения / Кабардинский Е.О., Ивашко А.Г.//Материалы Всероссийской научно-практической конференции молодых ученых с международным участием. В 2-х томах. Главный редактор А.П. Шкарапута. 2017. - С. 95 - 99.

5 Realistic data generator. [Электронный ресурс]. - https://mockaroo.com/ (Дата обращения: 03.05.2018)

6 SciKit learn. Multi-layer Perceptron regressor. [Электронный ресурс]. - http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html (Дата обращения: 03.05.2018)

7 Logic arrangement of services on the enterprise service bus (ESB). [Электронный ресурс]. - https://bluesoft.net.pl/en/logic-arrangement-of-services-on-the-enterprise-service-bus-esb/ (Дата обращения: 03.05.2018)

8 Виды Тестирования Программного Обеспечения. [Электронный ресурс]. Режим доступа: http://www.protesting.ru/testing/testtypes.html (Дата обращения: 04.06.2017)
9 Enterprise Integration Patterns. [Электронный ресурс]. Режим доступа: http://0agr.ru/blog/2012/08/08/enterprise-integration-patterns/ (Дата обращения: 25.12.2016)
10 Enterprise Integration Patterns. [Электронный ресурс]. Режим доступа: https://en.wikipedia.org/wiki/Enterprise_Integration_Patterns (Дата обращения: 25.12.2016)
11 Apache Camel. [Электронный ресурс]. Режим доступа: https://ru.wikipedia.org/wiki/Apache_Camel (Дата обращения: 25.12.2016)
12 Spring Boot [Электронный ресурс]. Режим доступа: http://spring-projects.ru/projects/spring-boot/ (Дата обращения: 25.12.2016)
13 ГОСТ Р 56920-2016/ISO/IEC/IEEE 29119-1:2013 Системная и программная инженерия. Тестирование программного обеспечения.
14 Guide to Software Engineering Body of Knowledge, SWEBOK, 2004
15 Орлик С. //Программная инженерия. Тестирование программного обеспечения
16 Docker. [Электронный ресурс]. Режим доступа: https://ru.wikipedia.org/wiki/Docker (Дата обращения: 25.12.2016)
17 Master/slave (technology). [Электронный ресурс]. Режим доступа: https://en.wikipedia.org/wiki/Master/slave_(technology) (Дата обращения: 04.06.2017)
18 Service-Oriented Architecture. [Электронный ресурс]. Режим доступа: http://www.service-architecture.com/articles/web- services/service-oriented_architecture_soa_definition.html (Дата обращения: 04.06.2017)
19 Introduction. [Электронный ресурс]. Режим доступа: https://maven.apache.org/what-is-maven.html (Дата обращения: 04.06.2017)

20 Введение в JSON. [Электронный ресурс]. Режим доступа: http://www.json.org/json-ru.html (Дата обращения: 04.06.2017)
21 Spring Boot. [Электронный ресурс]. Режим доступа: https://projects.spring.io/spring-boot/ (Дата обращения: 04.06.2017)
22 Gson User Guide [Электронный ресурс]. Режим доступа: https://github.com/google/gson/blob/master/UserGuide.md (Дата обращения: 04.06.2017)
23 Memory Management in the Java HotSpot™ Virtual Machine / Sun Microsystems//HotSpot™ Virtual Machine specification. 2006. – С.