РЕЧЬ

Слайд\_1: Добрый день, уважаемая комиссия. Сейчас я вам представлю свою выпускную квалификационную магистерскую работу на тему «Исследование методов межсервисного взаимодействия для передачи данных большого объема». Руководитель работы: кандидат технических наук Комоско Владимир Васильевич, рецензент – Теплых Никита Александрович, работу выполнил студент группы ПМ22М – Шаповал Ярослав Сергеевич.

Салйд\_2: В эпоху цифровизации и роста объемов данных, эффективность и скорость передачи данных становятся критически важными. С каждым днем количество информации, которую необходимо обрабатывать и передавать, увеличивается, что создает потребность в более эффективных методах передачи данных. Это особенно важно в контексте сервисно-ориентированных архитектур, где межсервисное взаимодействие является ключевым элементом.

Для быстрого обмена данными между сервисами требуется изучить современные методы передачи данных. Данная работа направлена на изучение и оптимизацию этого аспекта.

Слайд\_3: Целью данной работы являлось определение наиболее эффективного и быстрого метода передачи данных большого объема в сервисно - ориентированной архитектуре.

Для реализации поставленной цели, были выявлены следующие задачи:

* ﻿Изучение сервисно-ориентированных архитектур и протоколов НТТР/1.1 и НТТР/2.0.
* ﻿Разработка и реализация клиент-серверной системы на Python для передачи данных большого объема.
* ﻿Сравнение производительности различных методов передачи данных.

Слайд\_4: В данной работе объектом исследования определены методы межсервисного взаимодействия. А предметом исследования являлся алгоритм сравнения передачи данных большого объема в сервисно - ориентированной архитектуре.

Практическая значимость работы заключалась в использовании полученных результатов при разработке межсервисной системы взаимодействия.

Актуальность данной работы определена в ускорении межсервисного взаимодействия.

Слайд\_5: Сервисно - ориентированная архитектура (SOA) – это подход к разработке программного обеспечения, который позволяет создавать приложения в виде набора слабо связанных сервисов.

API – технология, позволяющая соединить функциональность разных компьютерных программ.

Для передачи данных по сети используются транспортные протоколы, особое внимание заслуживают НТТР/1.1 и его последняя версия – НТТР/2.0.

В сервисно - ориентированной архитектуре используются различные стили АРІ, каждый из которых имеет свои особенности, преимущества и недостатки.

Первый стиль о котором пойдет речь – SOAP. SOAP использует формат XML для запросов и ответов, что обеспечивает строгую типизацию, но может увеличить объем передаваемых данных. Он также поддерживает различные протоколы транспорта, включая НТТР, SMTP и даже JMS.

В отличие от SOAP, REST – следующая рассматриваемая архитектура, не имеет фиксированного формата. В нем можно обмениваться сообщениями на основе XML, JSON или любого другого удобного формата. Он использует стандартные НТТР - методы, такие как GET, POST, PUT и DELETE, для взаимодействия с ресурсами, что делает его простым и интуитивно понятным.

Третий стиль - GraphQL – это относительно новый стиль АРІ, который позволяет клиентам определять структуру запросов и ответов. Он позволяет клиентам запрашивать только те данные, которые им действительно нужны, что может существенно уменьшить объем передаваемых данных и улучшить производительность.

RPC (Remote Procedure Call) – это спецификация, которая позволяет удаленно выполнять функцию в другом контексте. RPC расширяет понятие локального вызова процедуры, но помещает его в контекст НТТР АРІ.

gRPC – новый и современный фреймворк для разработки масштабируемых, современных и быстрых АРІ, основанный на RPC, но уже в качестве транспорта использует новый НТТР/2. В нем для описания интерфейса используется Protocol Buffers, где описывается структура для передачи, кодирования и обмена данных. Он проще, компактнее и быстрее, чем XML, поскольку осуществляется передача бинарных данных.

Последняя рассмотренная архитектура – Socket. Это протокол, который обеспечивает двустороннее взаимодействие между клиентом и сервером в реальном времени. Socket может быть полезен для приложений, которым требуется непрерывное взаимодействие между клиентом и сервером.

Слайд\_6: Далее нужно рассказать о различных методах передачи данных большого объема в контексте клиент – серверной архитектуры. Начиная с синхронной передачи данных, переходя к многопоточности и более сложной асинхронной передаче данных, а после обсудить gRPC и сокеты.

Начну с синхронной передачи данных. В этом случае, сервер обрабатывает каждый запрос по очереди, блокируя выполнение до тех пор, пока не будет завершена обработка текущего запроса. Это означает, что во время обработки одного запроса, сервер не может начать обработку другого. Это является характерной особенностью синхронной передачи данных.

Слайд\_7: Следующие методы – многопроцессность и многопоточность.

Для начала нужно сказать про потоки и процессы. Программа, которая выполняется, называется процессом. Каждой программе выделяется область памяти не доступная другим процессам. Несколько процессов могут работать одновременно или параллельно используя ядра компьютера. В каждом процессе могут создаваться множество потоков, которые уже работают не параллельно, а конкурентно, то есть переходя от одного потока находящегося в ожидании к другому, требующему ресурсы. Это происходит настолько быстро, что пользователю порой кажется, что задачи выполнятся одновременно, но на самом деле они чередуются, и в каждый момент времени выполняется только одна задача. У потоков нет своей памяти, они пользуются памятью создавшего их процесса.

Многопроцессность в рамках данный работы рассматриваться не будет, так как применение этой технологии не дает преимуществ в IO – операциях, на которых основана данная работа.

Слайд\_8: Почему многопоточность используется при IO – операциях? Тут в силу вступает GIL. GIL не дает Python - процессу исполнять более одной команды байт – кода в каждый момент времени. Это означает, что, даже если имеется несколько потоков на многоядерной машине, интерпретатор сможет в каждый момент исполнять только один поток. Но GIL освобождается при вводе - выводе, как изображено на Рисунке 4. GIL захватывается снова, только когда полученные данные переносятся в объект Python.

К проблемам реализации многопоточности и обхода GIL можно отнести состояние гонки и взаимную блокировку.

Взаимная блокировка, это состояние, когда два или более потока находятся в состоянии ожидания ресурсов, занятых друг другом, и ни один из них не может продолжать свое выполнение.

Состояние гонки возникает, когда несколько потоков или процессов одновременно пытаются получить доступ к одним и тем же ресурсам без синхронизации. В результате, порядок, в котором операции выполняются, может влиять на конечный результат, что делает поведение программы неопределенным.

Слайд\_9: Следующий метод рассмотренный в данной работе – асинхронная передача данных. При работе с этим методом создаются сопрограммы. Сопрограммы – это облегченные потоки. Могут быть несколько сопрограмм работающих бок о бок в одном потоке. То есть получилась конкурентность в одном потоке.

Чтобы выполнить сопрограмму, нужно явно передать ее циклу событий. Цикл событий управляет очередью задач, как показано на Рисунке 5.

Задача – это обертка вокруг сопрограммы. Сопрограмма может приостановить выполнение, встретив операцию ввода - вывода, и дать циклу событий возможность выполнить другие задачи, которые не ждут завершения ввода - вывода.

Слайд\_10: Еще один способ передачи данных – использование низкоуровневых сокетов. Сокет – это низкоуровневая абстракция отправки и получения данных по сети. С их помощью производится обмен данными между клиентами и серверами.

Низкоуровневую концепцию сокетов проще понять, если представлять их как почтовые ящики. Можно положить письмо в почтовый ящик, почтальон заберет его и доставит в почтовый ящик получателя. Получатель откроет его и достанет данное письмо. В зависимости от содержания письма получатель может отправить ответ. Здесь письмо является аналогом данных или байтов, которые нужно отправить. Можно рассматривать помещение письма в почтовый ящик как запись байтов в сокет, а извлечение его из ящика – как чтение байтов из сокета, как показано на рисунке 6. Почтальон тогда является аналогом механизма передачи через интернет, который маршрутизирует данные до адреса назначения.

Сокеты условно можно разделить на клиентские и серверные. Серверные делают только прием запросов на установление соединения и создания клиентского сокета для каждого нового подключения. Клиентские сокеты не делают ничего, кроме как обмениваются данными. На рисунке 7 приведена схема, иллюстрирующая клиент - серверную модель взаимодействия. Суть ее в том, что разные клиенты подключаются к одному серверу, который координирует и управляет работой клиентов.

Слайд\_11: Последний рассмотренный вид передачи данных – gRPC. Это новый и современный фреймворк для разработки масштабируемых, современных и быстрых АРІ. Он позволяет писать микросервисы на любом языке, сохраняя при этом возможность легко устанавливать связь между этими сервисами. В качестве транспортного протокола используется новый НТТР/2. В gRPC можно определять свой АРI в терминах функций, а не НТТР - команд и ресурсов.

gRPC предоставляет четыре основных способа взаимодействия между клиентом и сервером, каждый из которых предназначен для различных сценариев использования.

При унарной передаче, продемонстрированной на рисунке 8 клиент отправляет один запрос и получает один ответ от сервера. Подходит для простых запросов и ответов, когда клиент ожидает результат в ответ на свой запрос.

Поток данных от сервера, продемонстрирован на рисунке 9. Клиент отправляет один запрос, а сервер отвечает потоком сообщений. Используется, когда клиенту нужно получить последовательность результатов от сервера, например, при потоковой передаче данных или мониторинге.

Слайд\_12: Поток данных от клиента – рисунок 10. Клиент отправляет поток сообщений, а сервер отвечает одним ответом. Применяется, когда клиент отправляет большой объем данных на сервер, но ожидает только один ответ, как в случае передачи файла.

Наконец, двунаправленный поток – рисунок 11. Клиент и сервер обмениваются потоками сообщений независимо друг от друга. Эффективно использовать, когда клиент и сервер могут отправлять данные асинхронно, что полезно для чат – приложений, потокового обмена данными и реактивных систем.

Слайд\_13: Реализовав на фреймворке FastApi описанные REST методы передачи данных были получены следующие результаты:

По протоколу HTTP 1 передача данных заняла примерно 8 секунд синхронным способом, 4.8 секунды асинхронным и 6 секунд используя многопоточность.

По протоколу HTTP 2 передача данных аналогичными методами составила примерно 40, 35 и 38 секунд соответственно. Это показывает то, что передача по HTTP 2 аналогична передаче по HTTP 1, но из – за проверки сертификатов безопасности на каждой итерации передачи занимает дольше времени.

По результатам видно, что передача данных асинхронным способом и в том и в другом протоколе самые быстрые.

Слайд\_14: Реализовав gRPC архитектуру тремя из четырех методов, были получены следующие результаты:

Унарная передача данных заняла примерно 7.7 секунд, потоковая передача данных от клиента – 5.3 секунд, двунаправленная передача данных 6.3 секунд.

По результатам видно что наилучшим способом себя показала потоковая передача от клиента, так как клиент получил ответ от сервера только после отправки всех данных.

Слайд\_15: Последний реализованный вид передачи данных – сокеты. Результаты сильно отличались в зависимости от выбора размера передаваемого количество байт за итерацию. Поэтому самым оптимальным и честным вариантом было использование динамического размера буфера, равного размеру файла. При такой передаче результат составил примерно 3.8 секунд.

Результат при альтернативном подходе с использованием статического размера буфера, с оптимально подобранными параметрами составил 3.1 секунды.

Слайд\_16: Чтобы сравнить полученные результаты я использовал процентное соотношение.

Асинхронная передача данных по HTTP 1.1 затратила на 40% меньше времени чем синхронная передача и на 20.55% меньше чем многопоточность. Самым долгим оказался метод с применением технологии UploadFile занявший на 291.7% времени больше чем синхронная передача, а это почти в 3 раза дольше.

Синхронная передача по протоколу HTTP/2 составляла 40.0234 секунд, что на 16.1% дольше чем асинхронная передача данных HTTP/2 и на 4.1% дольше многопоточности. Видно, что процентное соотношение между различными методами передачи данных примерно сохраняется для обоих протоколов.

Сравнение способов передачи данных с использованием gRPC показало, что потоковая передача от клиента оказалась самым быстрым методом, сокращая время передачи на 32.5% по сравнению с унарной передачей и на 14.3% по сравнению с двунаправленной передачей.

Слайд\_17: Самым быстрым вариантом из всех рассмотренных методов оказалась передача с использованием сокетов, с результатом в 3.866 секунд. Если взять его за 100%, то можно сравнить остальные методы относительно этого значения. Результаты сравнения отображены на Рисунке 19.

Исходя из расчетов, видно, что асинхронная передача данных по HTTP/1.1 занимает на 24.6% больше времени, чем передача данных с использованием сокетов. gRPC поток от клиента занимает – на 34.8% больше, а асинхронная передача по HTTP/2 значительно медленнее, занимая на 768.9% больше времени, то есть почти в 8 раз.

Слайд\_18: В заключении нужно сказать что поставленные задачи были выполнены. Произведено сравнение разработанных клиент – серверных архитектур. Из проведенного сравнения следует, что асинхронные методы передачи данных и многопоточность значительно превосходят синхронные методы по скорости передачи данных. Сокеты демонстрируют наилучшую производительность, особенно в сравнении с традиционными HTTP – методами. gRPC выделяется своей эффективностью, особенно при потоковой передаче данных от клиента, что делает его предпочтительным выбором для сценариев, требующих высокой производительности. HTTP/2, несмотря на свои теоретические преимущества, не показал ожидаемого улучшения производительности в данном исследовании, что может быть связано с дополнительной нагрузкой на обработку сертификатов безопасности. Знания полученные в результате проведенного в данной работе исследования можно применять в дальнейшей разработке сервисов на языке Python.