**АННОТАЦИЯ**

**Выпускная квалификационная работа содержит** \_\_ страниц, \_\_ рисунков, \_\_ таблиц и \_\_ источников.

**Ключевые слова**: межсервисная передача данных, протоколы передачи данных, сервисная архитектура, архитектуры передачи данных, скорость передачи, данные большого объема.

**Объектом исследования** являются методы межсервисного взаимодействия.

**Предметом исследования** является алгоритм сравнения передачи данных большого объема в сервисно-ориентированной архитектуре.

**Целью работы** является исследовать и сравнить методы передачи данных большого объема в сервисно-ориентированной архитектуре. (практически выяснить положительные и отрицательные стороны различных подходов передачи данных большого объема)

**Результатом работы** является практически доказанное сравнение передачи данных большого объема в сервисно-ориентированной архитектуре, обоснование возможности или невозможности, а также удобства использования каждого метода. (получить наилучший способ передачи, понять негативные стороны остальных методов)

Оглавление

[Список определений и сокращений 3](#_Toc166196505)

[Ведение 4](#_Toc166196506)

[ПОСТАНОВКА ЗАДАЧИ 7](#_Toc166196507)

[ГЛАВА 1 ОБЗОР СЕРВИСНО-ОРИЕНТИРОВАННЫХ АРХИТЕКТУР И ПРОТОКОЛОВ ПЕРЕДАЧИ ДАННЫХ ПО СЕТИ 7](#_Toc166196508)

[1.1 Что такое сервисы. Виды архитектур. 8](#_Toc166196509)

[1.2 Протоколы передачи данных. НТТР 1, НТТР 2 9](#_Toc166196510)

[1.3 Сравнение архитектурных стилей АРІ: SOAP, REST, GraphQL, RPC, WebSocket 10](#_Toc166196511)

[1.4 Вывод по главе 1 12](#_Toc166196512)

[ГЛАВА 2 МЕТОДЫ ПЕРЕДАЧИ ДАННЫХ БОЛЬШОГО РАЗМЕРА В КЛИЕНТ - СЕРВЕРНОЙ АРХИТЕКТУРЕ НА PYTHON 12](#_Toc166196513)

[2.1. FastApi 12](#_Toc166196514)

[2.\_ Сервера ASGI - uvicorn и hypercorn 13](#_Toc166196515)

[2.2 Синхронная передача данных 13](#_Toc166196516)

[2.3 Конкурентность и многопоточность 14](#_Toc166196517)

[2.4 Сокеты 18](#_Toc166196518)

[2.5 Uploadfile и его применение 24](#_Toc166196519)

[2.7 gRPC - Google Remote Procedure Call 25](#_Toc166196520)

[2.7.1 Использовать RPC и Protocol Buffers. 25](#_Toc166196521)

[2.7.2 Разновидности АРІ в gRPC 27](#_Toc166196522)

[2.6 Вывод по главе 2 28](#_Toc166196523)

[ГЛАВА З РЕАЛИЗАЦИЯ ВЫШЕОПИСАННЫХ МЕТОДОВ ПЕРЕДАЧИ ДАННЫХ В КЛИЕНТ - СЕРВЕРНОЙ АРХИТЕКТУРЕ 28](#_Toc166196524)

[3.1 Реализация передачи данных по НТТР1 28](#_Toc166196525)

[3.2 Реализация передачи данных по НТТР2 29](#_Toc166196526)

[3.3 Сравнение НТТР1 и НТТР2 29](#_Toc166196527)

[3.4 Вывод по главе 3 29](#_Toc166196528)

[ГЛАВА 4 СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ РАЗЛИЧНЫХ МЕТОДОВ ПЕРЕДАЧИ ДАННЫХ 29](#_Toc166196529)

[4.1 Методология сравнения 29](#_Toc166196530)

[4.2 Результаты передачи одного файла большого объема 29](#_Toc166196531)

[4.3 Результаты передачи множества файлов 29](#_Toc166196532)

[4.4 Выводы по главе 4 29](#_Toc166196533)

Список определений и сокращений

В таблице 1 приведены определения и сокращения, используемые в данной работе.

Таблица 1 – Используемые определения и сокращения

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Ведение

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. Использование современных алгоритмов открывает большое количество возможностей, в том числе способность работать в реальном времени. Данная работа направлена на изучение и оптимизацию этого аспекта, что делает ее актуальной и важной в современном мире.

Существуют различные исследования, посвященные передаче данных большого объема. Однако большинство из них сосредоточены на определенных типах данных или не учитывают специфику сервисно-ориентированных архитектур. Кроме того, многие из них не рассматривают различные сценарии передачи данных. Эта работа направлена на заполнение этих пробелов. В ней рассматриваются различные сценарии передачи данных, для поиска наиболее эффективного метода для каждого сценария. Это делает данную работу уникальной и важной для текущего исследовательского контекста.

В данной работе объектом исследования является система передачи данных большого объема в сервисно-ориентированной архитектуре.

Предметом исследования является анализ и сравнение методов передачи больших объемов данных в сервисно-ориентированной архитектуре.

Целью работы является определение наиболее эффективного и быстрого метода передачи данных большого объема в сервисно-ориентированной архитектуре.

Для достижения поставленной цели решались следующие задачи:

* ﻿﻿Изучение сервисно-ориентированных архитектур и протоколов НТТР 1 и НТТР 2.
* ﻿﻿Разработка и реализация клиент-серверной системы на Python для передачи данных большого объема.
* ﻿﻿Сравнение производительности различных методов передачи данных.

При проведении исследований использовались следующие методы:

* ﻿﻿Анализ источников, находящихся в свободном доступе, описывающих работу приложений с сервисно - ориентированной архитектурой
* ﻿﻿Анализ и сравнение различных методов передачи данных.
* ﻿﻿Разработка и тестирование клиент-серверной системы, проверка работы различных подходов и протоколов.
* Использование синхронной и асинхронной передачи данных в FastApi, потоков, gRPC и веб-сокетов для передачи данных.
* ﻿﻿Процентное соотношение и визуальное сравнение результатов

Научной новизной обладают следующие аспекты работы:

* ﻿﻿Исследование и сравнение производительности различных методов передачи данных в сервисно-ориентированных архитектурах.
* ﻿﻿Рассмотрение различных протоколов передачи, рассмотрение передачи данных в общем виде.

Практическая значимость работы: Результаты этой работы могут быть использованы для оптимизации передачи данных в сервисно-ориентированных архитектурах. Они могут помочь разработчикам выбрать наиболее эффективный метод передачи данных, что в свою очередь может улучшить производительность и надежность их систем.

Основная часть магистерской диссертации состоит из следующих разделов:

1. ﻿﻿﻿Обзор сервисно-ориентированных архитектур и протоколов НТТР 1 и НТТР 2.
2. ﻿﻿﻿Методы передачи данных большого объема клиент - серверной системы на Python.
3. ﻿﻿﻿Реализация вышеупомянутых методов.
4. ﻿﻿﻿Сравнение производительности различных методов передачи данных и выводы.
5. ﻿﻿﻿Комплекс мероприятий по охране труда для обеспечения безопасных условий на рабочем месте.

ПОСТАНОВКА ЗАДАЧИ

В данной работе необходимо изучить сервисно - ориентированную архитектуру, протоколы передачи данных. После этого исследовать методы передачи данных в рассмотренной архитектуре. Также ее нужно реализовать и протестировать на различные методы передачи больших данных.

Необходимо проделать вышеперечисленную работу для двух видов передаваемых данных - большого количества файлов маленького размера и одного файла большого размера. Для достижения поставленной цели необходимо произвести сравнение результатов, сделать выводы.

В рамках выполнения данной работы были поставлены следующие

задачи:

1. ﻿﻿﻿Изучение сервисно - ориентированной архитектуры и протоколов передачи данных.
2. ﻿﻿﻿Исследование существующих подходов взаимодействия для передачи данных большого объема в сервисной архитектуре.
3. ﻿﻿﻿Разработка алгоритма взаимодействия.
4. ﻿﻿﻿Проведение исследований для оценки скорости передачи каждого метода.

# ГЛАВА 1 ОБЗОР СЕРВИСНО-ОРИЕНТИРОВАННЫХ АРХИТЕКТУР И ПРОТОКОЛОВ ПЕРЕДАЧИ ДАННЫХ ПО СЕТИ

В данной главе вводятся понятия, используемые в дальнейшем. Приводится обзор литературы по теме клиент - серверной архитектуры взаимодействия.

## 1.1 Что такое сервисы. Виды архитектур.

Сервисы в контексте программного обеспечения - это отдельные функциональные единицы, которые могут быть использованы в различных приложениях и бизнес-процессах. Сервисы обычно предоставляют определенный набор функций и взаимодействуют друг с другом через определенные протоколы и интерфейсы.

Сервисно - ориентированная архитектура (SOA) - это подход к разработке программного обеспечения, который позволяет создавать приложения в виде набора слабо связанных сервисов. В SOA, сервисы обычно взаимодействуют друг с другом посредством протоколов обмена сообщениями.

Существуют различные виды архитектур, включая монолитную, микросервисную, и серверную архитектуру.

Монолитная архитектура представляет собой традиционный подход к разработке программного обеспечения, где все компоненты приложения объединены в один модуль или единицу. Это может упростить разработку и тестирование, но может привести к проблемам с масштабируемостью и гибкостью приложения.

Микросервисная архитектура - это подход, при котором приложение строится как набор небольших сервисов, каждый из которых выполняет конкретную бизнес-функцию и может разрабатываться, развертываться и масштабироваться независимо. Это может улучшить гибкость и масштабируемость приложения, но также может увеличить сложность управления и координации между сервисами.

Серверная архитектура - это подход, при котором логика приложения разделена между сервером и клиентом. Сервер обычно отвечает за обработку данных и бизнес-логику, в то время как клиент отвечает за представление данных пользователю. Это может улучшить производительность и эффективность приложения, но также может увеличить сложность управления и синхронизации между сервером и клиентом.

Каждый из этих подходов имеет свои преимущества и недостатки, и выбор между ними обычно зависит от конкретных требований и контекста приложения. В контексте данной работы, заострено внимание на сервисно-ориентированной архитектуре и изучаются методы передачи данных в таких системах.

Существует большое количество различных способов передачи информации между сервисами. Все они основаны на различных подходах к хранению и обмену данными. Для лучшего понимания стоит более детально рассмотреть протоколы, среди которых особое внимание заслуживают НТТР и его последняя версия - НТТР/2.

## 1.2 Протоколы передачи данных. НТТР 1, НТТР 2

НТТР/1.1 – это\_\_\_\_ . Он поддерживает множество запросов через одно соединение, дает возможность отправлять несколько запросов без ожидания ответов, также позволяет виртуальным хостам существовать на одном сервере, но он все еще подвержен проблемам блокировок из-за одного соединения, особенно при множестве мелких файлов.

НТТР/2 был представлен в 2015 году для улучшения производительности. Множество запросов может выполняться параллельно в рамках одного соединения, разделенного на потоки. Его плюсами также является уменьшение объема передаваемых данных, возможность устанавливать приоритеты запросов, возможность сервера предварительно отправлять ресурсы, которые могут быть запрошены клиентом. В целом НТТР/2 предоставляет значительное улучшение производительности за счет множественных потоков, сжатия заголовков и других оптимизаций. Он особенно эффективен при загрузке больших веб-страниц и ресурсов, снижая задержки и ускоряя время загрузки.

Программы обычно сталкиваются с данными в различных форматах: данные, хранящиеся в памяти, и данные, предназначенные для записи в файлы и передачи по сети. Обычные структуры данных в памяти оптимизированы для эффективного взаимодействия с центральным процессором. Однако, для передачи по сети или записи в файл, эти данные необходимо преобразовать в последовательность байтов. Этот процесс называется кодированием или сериализацией. Разнообразные форматы, такие как JSON, XML и CSV, используются для представления данных в виде последовательности байтов.

Существует несколько способов организации взаимодействия процессов по сети, включая SOAP, REST, GraphQL, RPC и WebSocket.

## 1.3 Сравнение архитектурных стилей АРІ: SOAP, REST, GraphQL, RPC, WebSocket

API (Application Programming Interface) - технология, позволяющая соединить функциональность разных компьютерных программ. АРІ можно сравнить с официантом, который получает от клиента заказ из ограниченных пунктов меню, передает его на кухню («системе»), а после приготовления, возвращает готовые «блюда» заказчику.

Почти все, с чем можно столкнуться в интернете, имеет отношение к

API, а точнее к версиям этого программного интерфейса, использующим для работы НТТР-запросы. Когда пользователь хочет узнать прогноз погоды, интерфейс браузера или мобильного приложения вызывает АРІ Яндекс.Погоды или API Gismeteo, когда нужно проложить кратчайший маршрут из одного места в другое, Яндекс.Карты вызывают соответствующее АРТ.

Пользовательские АРІ-интерфейсы могут быть реализованы на Python с использованием нескольких фреймворков. В этой статье остановимся на особенностях работы с одним из самых популярных вариантов - платформой FastAPI.

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

SOAP (Simple Object Access Protocol) - это протокол обмена структурированными сообщениями в распределенной вычислительной среде. SOAP был разработан для обеспечения взаимодействия между объектами по сети, в основном для реализации удаленного вызова процедур (RPC). 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, что делает его универсальным решением для различных сценариев использования.

REST (Representational State Transfer) - это архитектурный стиль, который использует принципы НТТР для создания АРІ. В отличие от SOAP, который использует формат XML для запросов и ответов, REST 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 для разработчиков.

GraphQL - это относительно новый стиль АРІ, который позволяет клиентам определять структуру запросов и ответов. Это может упростить интеграцию и улучшить производительность за счет уменьшения количества данных, передаваемых по сети. GraphQL 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. Он может быть реализован с использованием различных протоколов, включая

SOAP и JSON-RPC, и может поддерживать различные стили взаимодействия, включая запрос-ответ, уведомления и даже потоковую передачу данных.

WebSocket - это протокол, который обеспечивает двустороннее взаимодействие между клиентом и сервером в реальном времени. WebSocket может быть полезен для приложений, которым требуется непрерывное взаимодействие между клиентом и сервером, таких как чаты, игры в реальном времени и другие интерактивные приложения.

## 1.4 Вывод по главе 1

В данной главе приведен краткий обзор клиент - серверного взаимодействия, описаны виды архитектурных стилей построения API, а так же протоколы передачи данных. В следующей главе будут подробно описаны методы передачи данных большого объема в выбранной архитектуре.

# ГЛАВА 2 МЕТОДЫ ПЕРЕДАЧИ ДАННЫХ БОЛЬШОГО РАЗМЕРА В КЛИЕНТ - СЕРВЕРНОЙ АРХИТЕКТУРЕ НА PYTHON

Данная глава посвящена различным методам передачи данных большого объема в контексте клиент - серверной архитектуры, реализованной на языке программирования Python. Начиная с обзора FastAPI, погружаясь в синхронную и асинхронную передачу данных, выделяются преимущества от передачи данных в виде байтов и недостатки использования технологии Uploadfile. Также в главе 2 подробно описываются такие технологии как gRPC, многопроцессорные и много потоковые подходы, сокеты и передача данных с их помощью.

## 2.1. FastApi

2.1.1 FastAPI - является легковесным асинхронным фреймворком для

Python, который используют преимущественно для разработки АРI-сервисов.

FastAPI обходит все Python-фреймворки по производительности. Также к плюсам разработки на FastAPI можно отнести скорость работы, гибкость, автоматическая OpenAPl документация, асинхронность. Все это дает возможность эффективно обрабатывать большое количество запросов параллельно. Стоит отметить встроенную валидацию данных в FastAPI -

Pydantic. Вместо того, чтобы вручную писать проверки для содержимого json-файлов и обрабатывать ошибки валидации, он автоматизирует эти процессы и ускоряет валидацию данных в 20 раз.

Из минусов данного фреймворка можно выделить ограниченное

количество понятного обучающего материала, сложность вхождения в тему асинхронной разработки.

Остановив выбор на использовании FastAPI для реализации клиент - серверного приложения, ниже будут рассмотрены различные способы передачи данных.

## 2.\_ Сервера ASGI - uvicorn и hypercorn

ASGI — Asynchronous Standard Gateway Interface. Верхнеуровнево ASGI можно рассматривать как связующее звено, которое позволяет асинхронным Python серверам и приложениям взаимодействовать друг с другом. Вот так его можно изобразить на диаграмме:



Рисунок\_\_ -

## 2.2 Синхронная передача данных

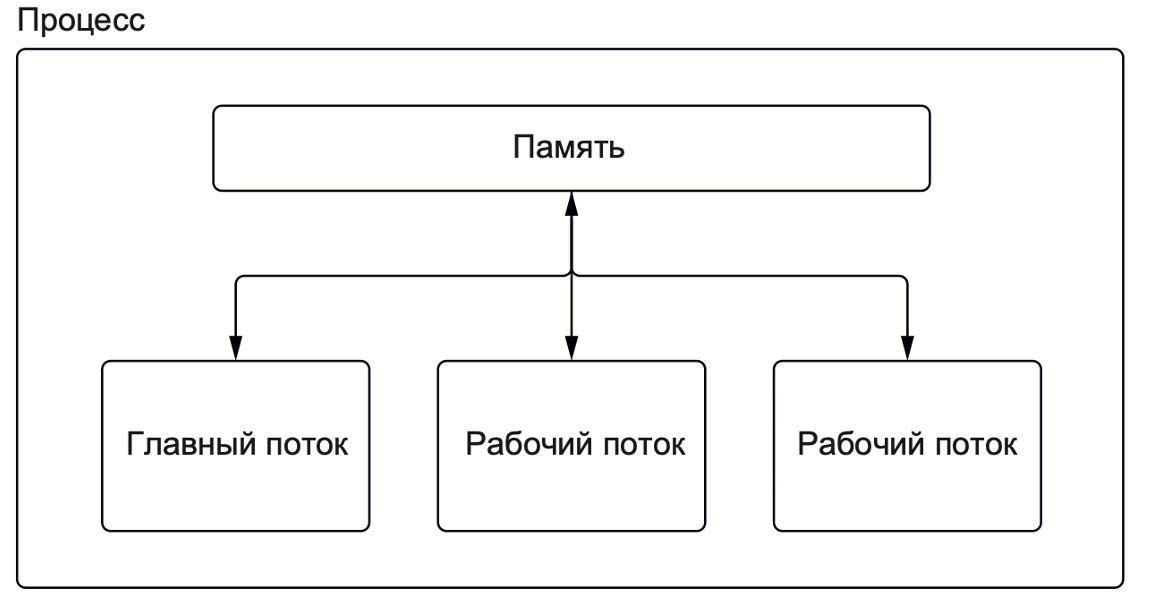
Синхронная передача данных - это метод передачи, при котором отправитель и получатель данных синхронизируются друг с другом. Это означает, что получатель должен быть готов принять данные в тот момент, когда отправитель готов их отправить.

В контексте клиент-серверной архитектуры, синхронная передача обычно означает, что клиент (или отправитель данных) ожидает ответа от сервера (или получателя данных) перед тем, как продолжить свою работу, и не может выполнять другие действия. Клиент блокируется или ожидает, пока сервер не обработает запрос и не вернет ответ. В FastAPI, синхронная передача данных может быть реализована с использованием обычных функций Python. Она может быть проста в реализации, но не неэффективной, если обработка запроса занимает много времени, поскольку клиент должен ожидать окончания обработки, прежде чем он сможет продолжить свою работу. Это может привести к задержкам и снижению производительности, особенно при обработке больших объемов данных или при выполнении сложных операций. По этой причине в некоторых случаях может быть предпочтительнее использовать асинхронную передачу данных, к которой постепенно будем приближаться в следующих разделах.

## 2.3 Конкурентность и многопоточность

Программа — это определенная последовательность инструкций и данных, решающих определенную задачу, которая хранится в памяти. Программа, которая выполняется, называется процессом. Каждой программе выделяется ячейка памяти не доступная другим **процессам**. Самый главный процесс — это операционная система (ОС). Она в свою очередь может запускать другие программы как процессы ОС. Если машина оснащена процессором с несколькими ядрами, то несколько процессов могут работать одновременно или **параллельно**. То есть выполняться строго в одно и то же время. Если процессор имеет только одно ядро, то все равно можно выполнять несколько приложений **конкурентно**, но уже с применением квантования времени, при котором операционная система будет автоматически вытеснять работающий процесс по истечении некоторого промежутка времени и передавать процессор другому процессу. 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. Программы, которые являются частью ОС, называются системными. Все остальные - прикладными, или приложениями. Они запускаются пользователем и предназначены для решения его задач. Программа может сама управлять только ходом собственного исполнения (инструкции if, else, while, for и так далее) и вычислениями. Всё же остальное - выделение блока памяти, вывод на экран, обращение к мыши и клавиатуре, сети, файлам, жестким дискам и другим устройствам ввода-вывода - программа может получить, только обратившись к ОС через системный вызов. Чаще всего процессы применяют для вычислительных задач. С каждым процессом создается как минимум один **поток** - главный. У потоков нет своей памяти, они пользуются памятью создавшего их процесса. Дополнительные потоки, созданные процессом называются рабочими или фоновыми потоками. Они выполняют работу конкурентно - быстро переключаясь с одного на другой и используют общую память создавшего их процесса. Потоки в основном используют при выполнении IO - операций - операций, ограниченных производительностью ввода-вывода, например запросы к серверу или базе данных. Пока выполняется что-либо не зависящее от нас, мы переходим к выполнению следующей задачи, в то время как предыдущая задача ожидает ответа.

Подытоживая можно выделить два вида работы – выполнение нескольких задач параллельно за счет ядер машины и выполнение задач конкурентно, переключаясь между IO - задачами находящимися в ожидании, то есть **многопроцессность** и **многопоточность**.



Рисунок\_\_\_ - Многопоточная программа с двумя рабочими потоками и одним главным.

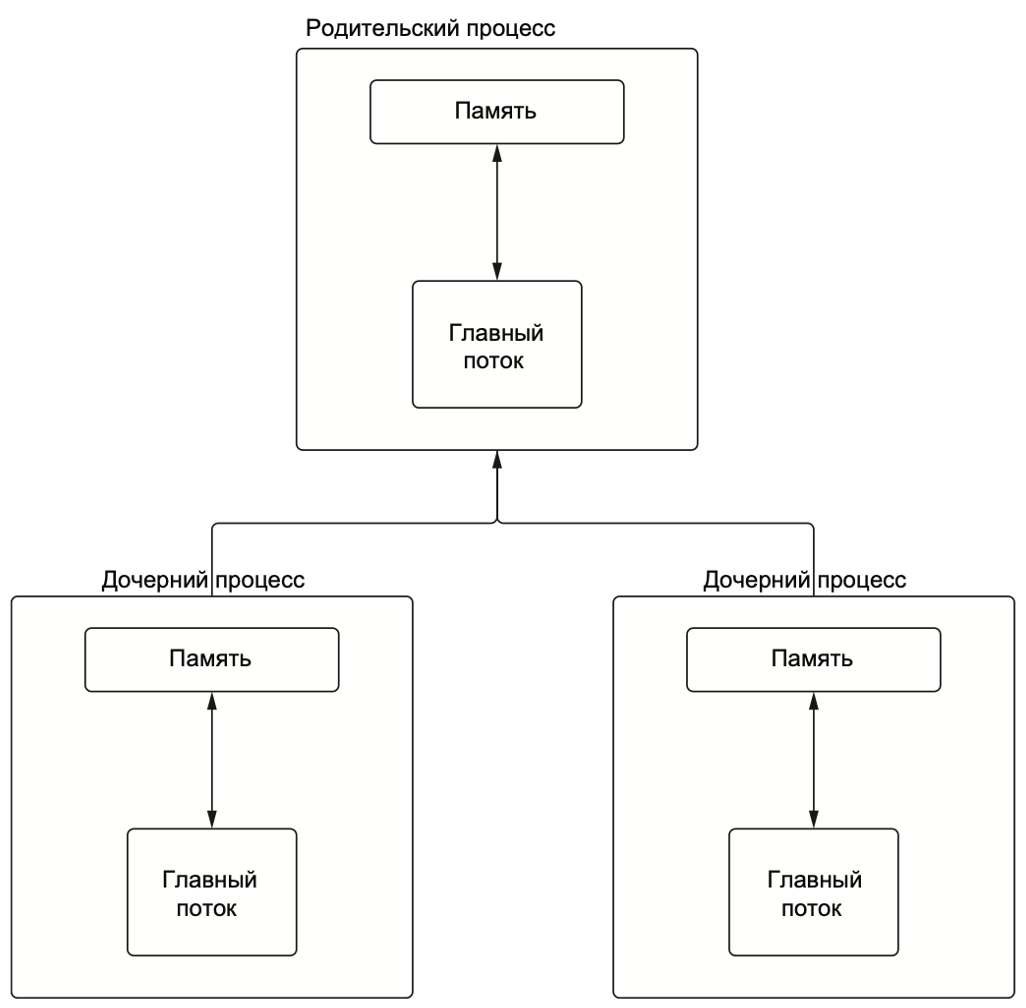


Рисунок \_\_ - Приложение в котором имеется один родительский процесс и два дочерних.

Многопроцессность в рамках данный работы рассматриваться не будет.

Для работы с многопоточностью в Python используются такие библиотеки, как, например, Threads или concurrent.futures. Создание потока в Python осуществляется путем создания экземпляра класса Thread и передачи целевой функции (функции, которую выполнит поток) в качестве аргумента.

Но со временем был создан еще более удобный и быстрый вариант -конкурентность в одном потоке. В Python это реализовано в библиотеке

Asyncio. Прежде чем перейти к асинхронному программированию стоит упомянуть главную проблему параллельного программирования на Python - Глобальная блокировка интерпретатора (global interpreter lock - 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. GIL освобождается при вводе-выводе. В этом случае низкоуровневые системные вызовы работают за пределами среды выполнения Python и не зависят от него. GIL захватывается снова, только когда полученные данные переносятся в объект Python.

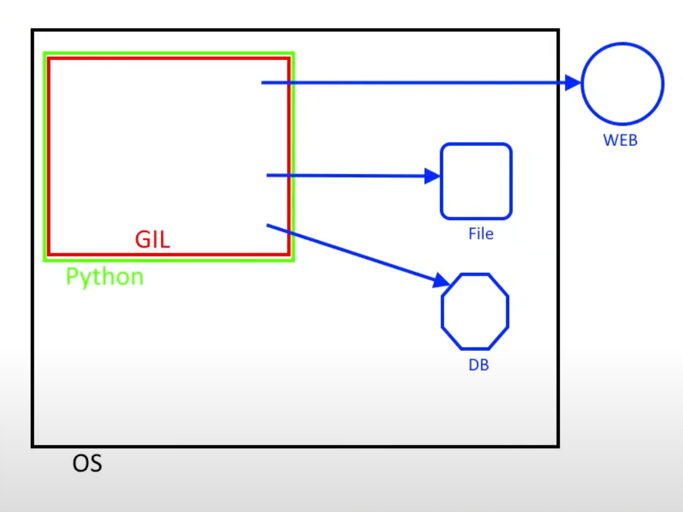


Рисунок \_ - Схема влияния GIL и конец его владений

При работе с Asyncio создаются сопрограммы. **Сопрограммы** – это облегченные потоки. Может быть несколько сопрограмм работающих бок о бок в одном потоке. Но Asyncio все еще не обходит GIL. Также, если имеется счетная задача, то для ее конкурентного выполнения все равно нужно заводить отдельный процесс. Сопрограммы не выполняются, если их вызвать напрямую. Чтобы выполнить сопрограмму, нужно явно передать ее циклу событий.

Как отличить задачи ожидающие завершения ввода - вывода от тех, которые просто выполняют python код и ничего не ждут? Решение - уже ранее упомянутая конструкция **event - loop (цикл событий)**

B asyncio цикл событий управляет очередью задач. 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.

Создавая цикл событий, мы создаем пустую очередь задач. Затем добавляем в эту очередь задачи для выполнения. На каждой итерации цикла проверяется, есть ли в очереди готовая задача, и если да, то она выполняется, пока не встретит операцию ввода-вывода. В этот момент задача приостанавливается, и мы просим операционную систему наблюдать за ее сокетами. А сами тем временем переходим к следующей готовой задаче. На каждой итерации проверяется, завершилась ли какая-нибудь операция ввода-вывода; если да, то ожидавшие ее завершения задачи пробуждаются и им предоставляется возможность продолжить работу.

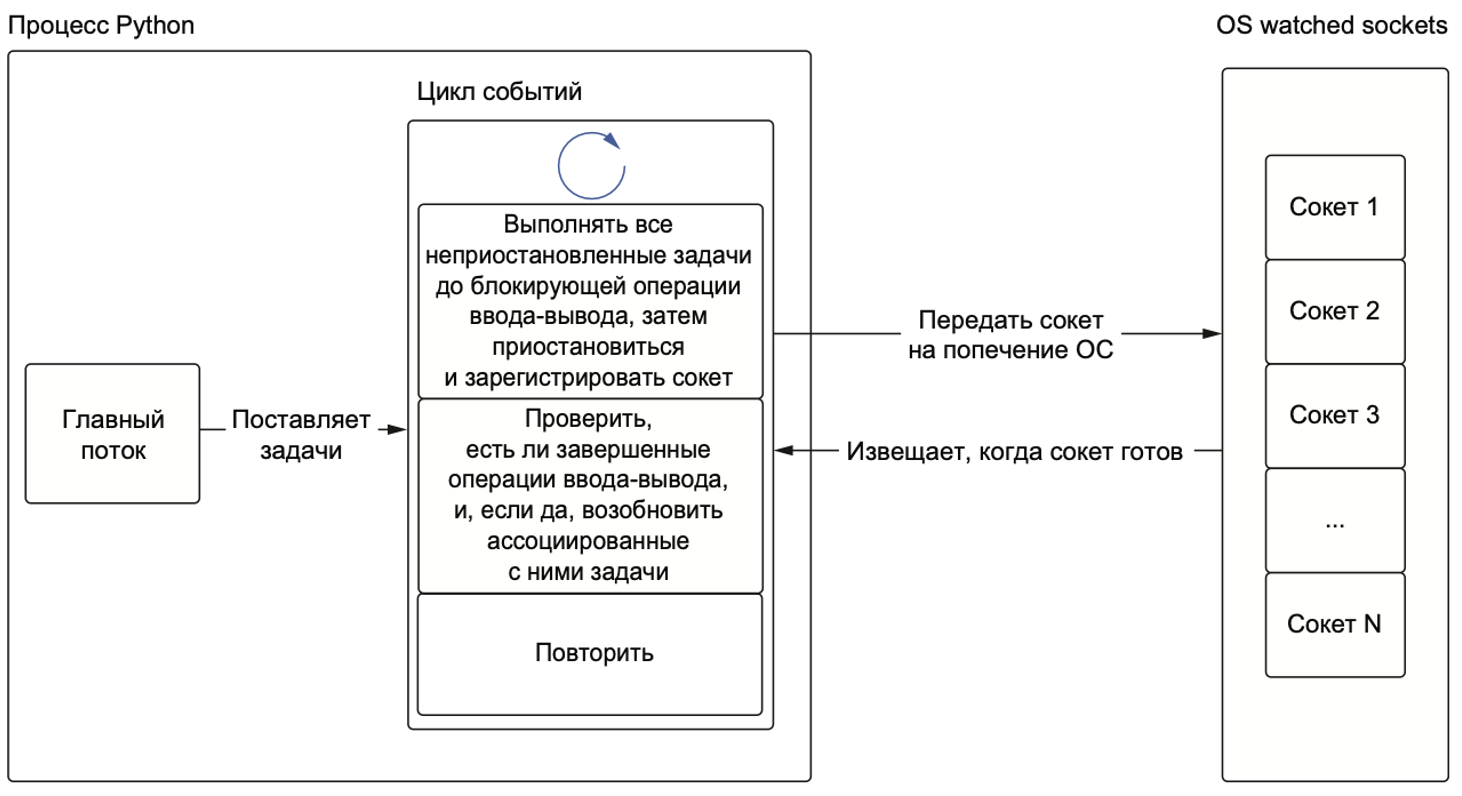


Рисунок \_\_ - Пример потока поставляющего задачи циклу событий.

Каждый момент времени работает только один счетный кусок кода, но при этом конкурентно выполняются одна или две операции ввода-вывода, как показано на рисунке \_\_. Именно из-за такого перекрытия ожидания ввода-вывода asyncio и достигает экономии во времени.

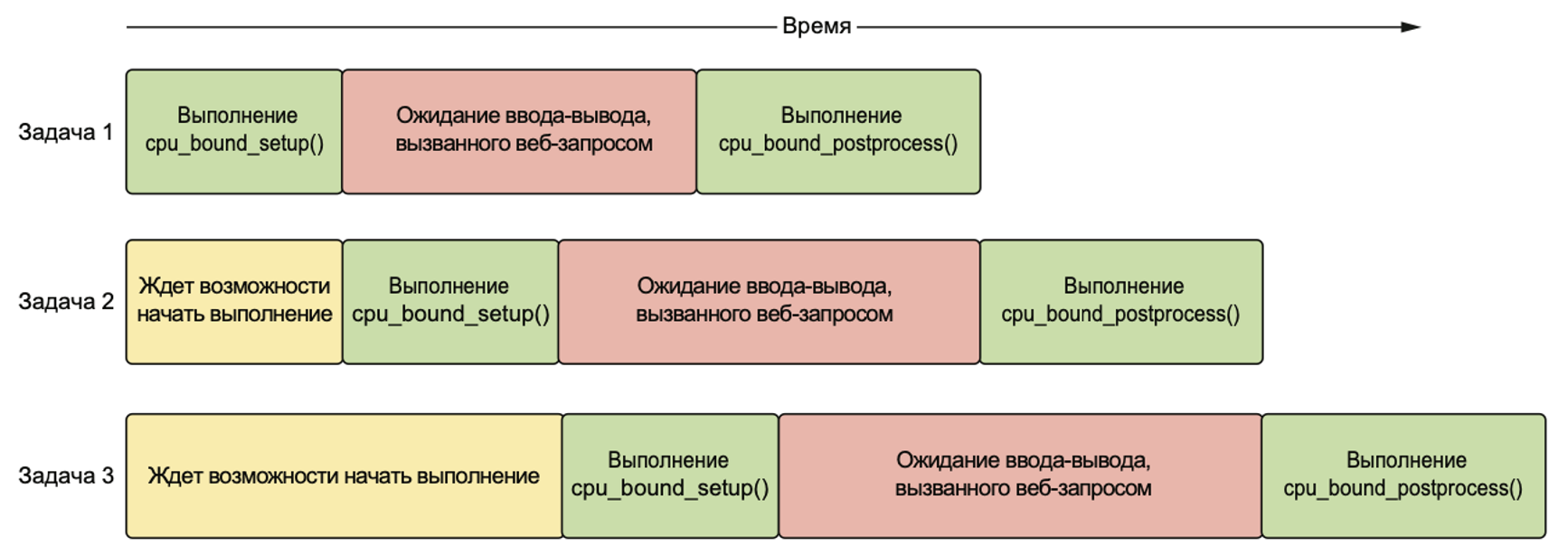
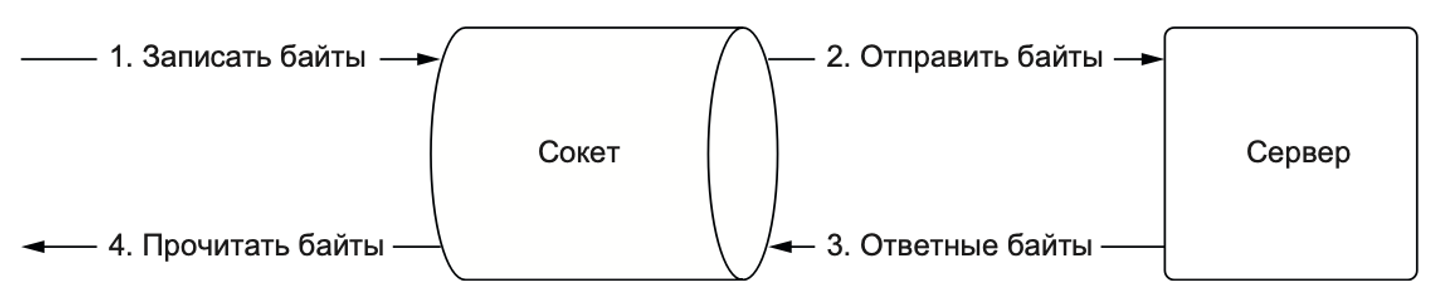


Рисунок \_\_ - Конкурентное выполнение нескольких задач с операциями ввода – вывода.

## 2.4 Сокеты

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. Вы можете положить письмо в почтовый ящик, почтальон заберет его и доставит в почтовый ящик получателя. Получатель откроет его и достанет ваше письмо. В зависимости от содержания письма получатель может отправить в ответ. Здесь письмо является аналогом данных или байтов, которые мы хотим отправить. Можно рассматривать помещение письма в почтовый ящик как запись байтов в сокет, а извлечение его из ящика - как чтение байтов из сокета. Почтальон тогда является аналогом механизма передачи через интернет, который маршрутизирует данные до адреса назначения.



Рисунок\_\_ - Запись байтов в сокет и чтение байтов из сокета

Сокеты работают на транспортном уровне протоколов и соответственно бывают 2 типов:

Потоковые (на основе ТСР) - сокеты с установленным соединением на основе протокола ТСР, передают поток байтов, который может быть двунаправленным - т.е. приложение может и получать и отправлять данные.

Дейтаграммные (на основе UDP)- сокеты, не требующие установления явного соединения между ними. Сообщение отправляется указанному сокету и, соответственно, может получаться от указанного сокета.

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. Ниже на рисунке\_\_ приведена схема, иллюстрирующая клиент-серверную модель взаимодействия. Суть ее в том, что разные клиенты подключаются к одному серверу, который координирует и управляет работой клиентов.

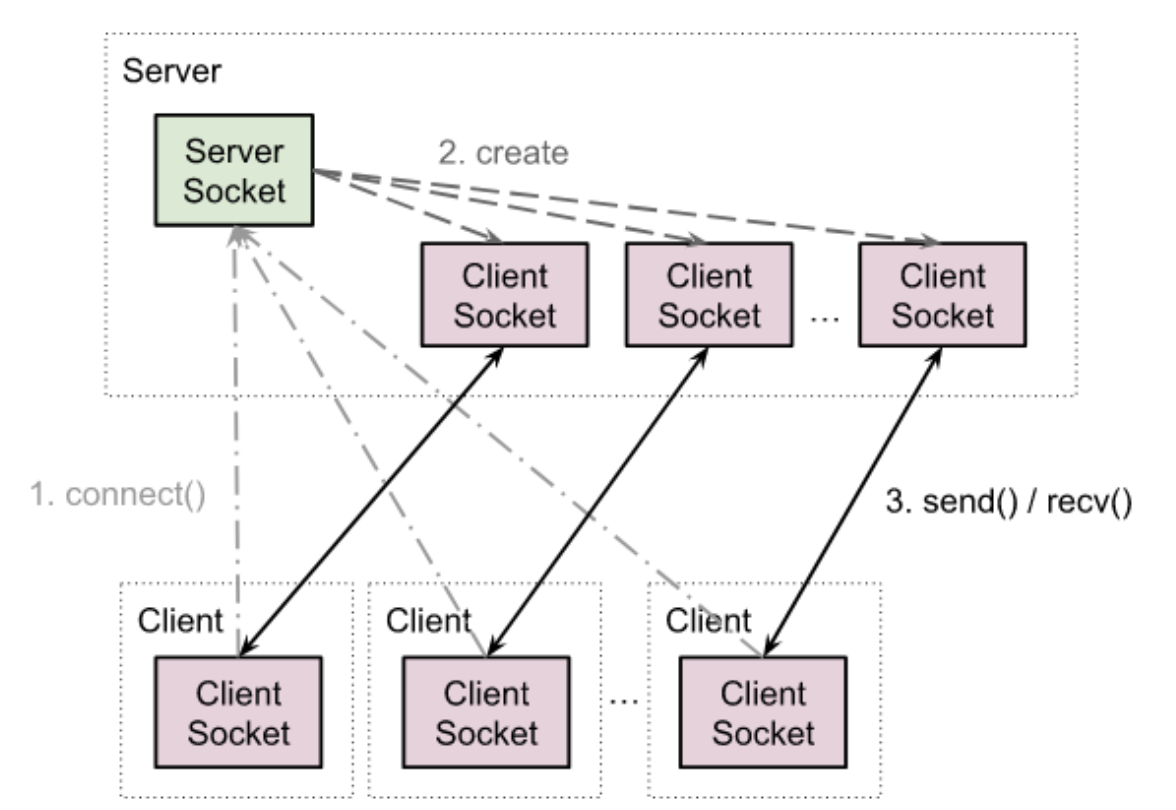


Рисунок \_\_- Схема взаимодействия клиентских сокетов.

Сокет состоит из Р-адреса и порта.

ІР-адрес [HOST] - уникальный сетевой адрес узла в компьютерной сети, построенной по протоколу IP. В версии протокола IPv4 Р-адрес имеет длину 4 байта (например, 192.168.0.3), а в версии протокола IPv6 IP-адрес имеет длину 16 байт (например, 2001:0db8:85а3:0000:0000:8a2e:0370:7334).

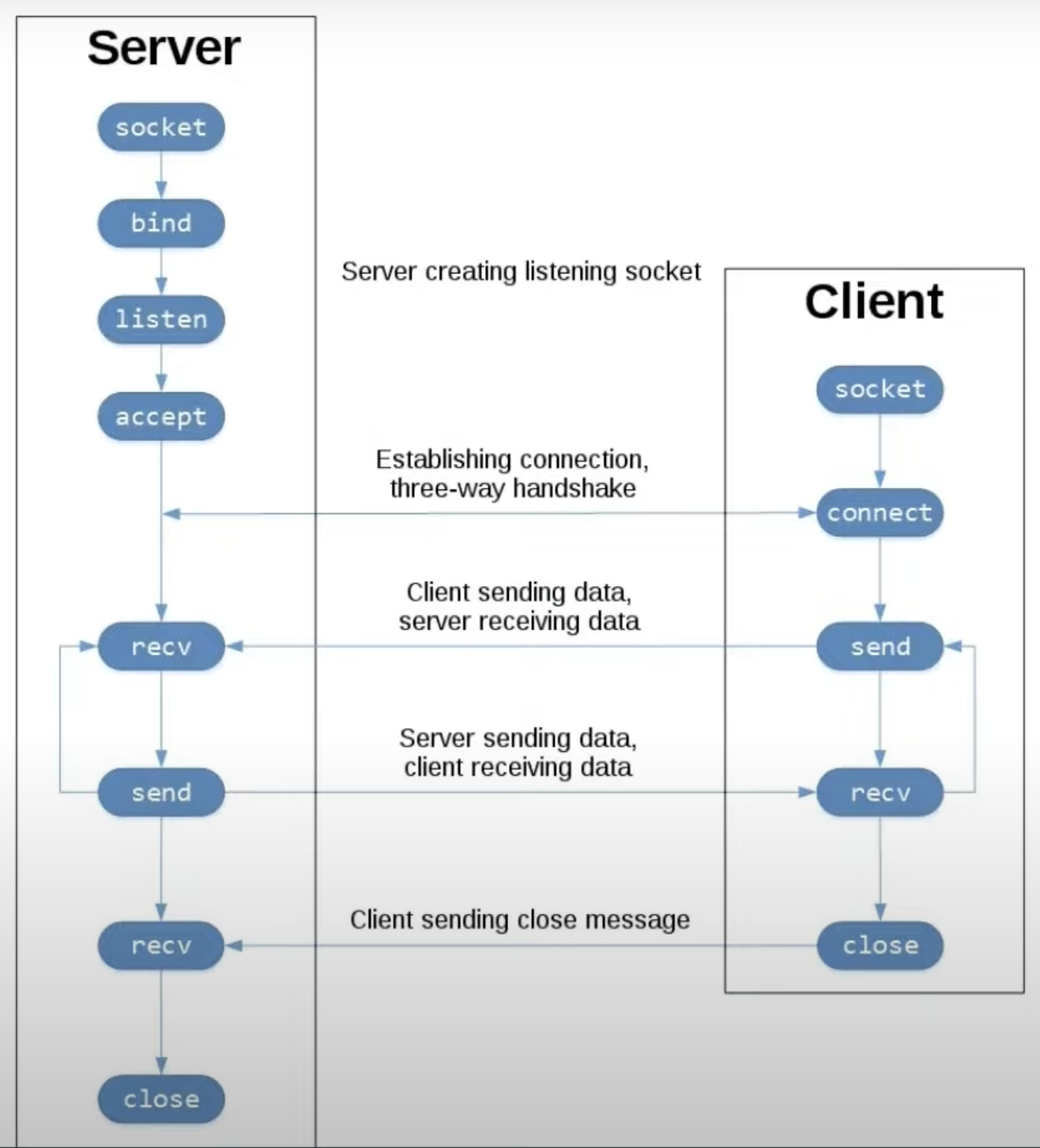
IР-адрес должен быть уникален. Если сокет должен быть виден только снаружи, из других систем, то нужно взять значение из server\_sock.gethostname(). Если он должен быть виден только изнутри, то необходимо задать "localhost" или "127.0.0.1". Значение "" осуществляет привязку ко всем доступным для данной машины адресам: и локальным, и глобальным. Заметим, что использование "localhost" для локального доступа позволяет пропустить несколько слоев сетевого кода, что быстрее, чем использовать HOST="".

Порт - натуральное число, записываемое в заголовках протоколов транспортного уровня (ТСР, UDP и др.). Порт используется для определения процесса-получателя пакета в пределах одного хоста. Так как малые значения (до 1024) большей частью зарезервированы, то рекомендуется использовать любое 4-значное число свыше 1024.

Основные функции объекта socket:

* .socket() - Создаётся объект сокета
* .bind(address) - Привязывает сокет к адресу address (инициализирует IP-адрес и порт). Сокет не должен быть привязан до этого.
* .listen() - Переводит сервер в режим приема соединений.
* .accept() - Принимает соединение и блокирует приложение в ожидании сообщения от клиента.
* .send() - Отправляет данные клиенту и возвращает количество отправленных байтов. Нужно самостоятельно управлять повторными попытками отправки данных, если они дошли не все. Переданная в аргументе последовательность байтов добавляется в буфер отправки для соответствующего соединения. Когда именно буфер будет отправлен, зависит от операционной системы и уровня ее загруженности.
* .sendall(data) - Гарантирует отправку всех данных, вызывает низкоуровневый s.send(data) пока не будет отправлено все.
* .recv() - Читает и возвращает данные в двоичном формате (набор байтов) из сокета. В методе .recv() мы передаем максимальное число байтов, которое мы желаем получить. Если в буфере меньше данных, чем мы запрашиваем, то функция возвращает, сколько есть. Лучше всего, если число будет степенью двойки и не меньше максимальной ожидаемой длины сообщений.
* .close() - Закрывает сокет.
* Метод encode('utf-8') - позволяет перевести данные из строки в байты.
* Метод decode('utf-8') - позволяет перевести данные из байт в строку.

Полный же жизненный цикл сокета завершается с закрытием соединения как показано на рисунке\_



Рисунок\_\_ - Схема полного пути взаимодействия клиента и сервера

Методы accept(), connect() и recv() класса socket являются блокирующими. Это значит, что после их вызова выполнение программы остановится и не возобновится до тех пор, пока не поступят соответствующие данные из сети. А если программа не выполняется, то и другие соединения и задачи также не обрабатываются. Получается что тут происходит синхронная передача данных, аналогично тому как было описано в пункте 2.2.

Сокеты могут работать в неблокирующем режиме, когда мы просто начинаем операцию чтения или записи и забываем о ней, а сами занимаемся другими делами. Но позже операционная система уведомляет нас о том, что байты получены, и в этот момент мы можем уделить им внимание. Для реализации такой схемы применяются различные системы уведомления с помощью событий, разные в разных ОС.

В Python есть несколько способов реализовать многозадачность:

* ﻿Процессы (создаются копии самого процесса приложения) multiprocessing.
* Потоки (выполняются в одном процессе) multithreading.
* ﻿﻿Системный вызов select и его аналоги (выполняются в одном потоке).

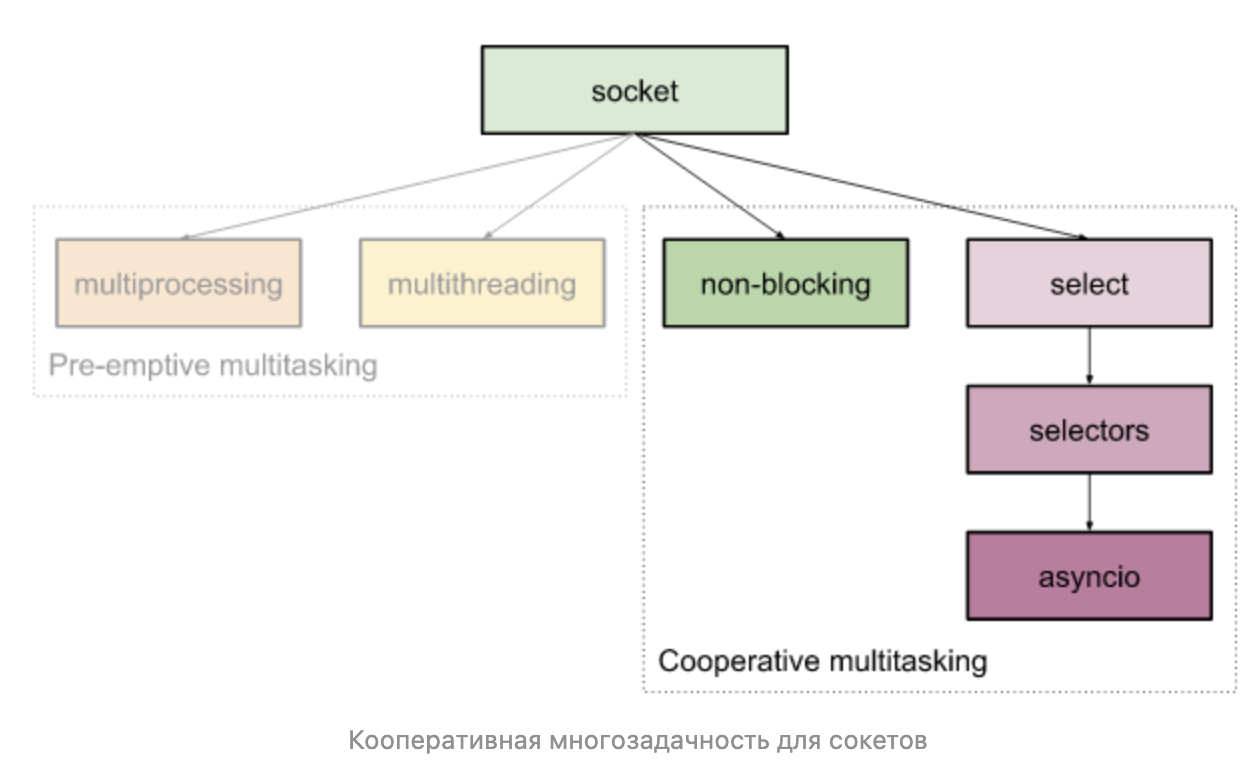


Рисунок \_\_- Вытесняющая и кооперативная многозадачность для сокетов.

Мультиплексирование с помощью select. Использование системного вызова select означает что переключение между задачами осуществляется вручную и полностью управляется из кода. Никаких неожиданных перескакиваний из задачи в задачу, как это было в потоках или процессах, нет. Поэтому здесь не нужно никакой синхронизации и защиты от взаимных блокировок. Весь код выполняется синхронно — инструкция за инструкцией.

Принцип работы функции select() довольно простой — передавая в select() список всех сокетов, мы на выходе получим список тех из них, которые готовы к обработке. В соответствии с полученным сокетом мы сами выбираем, какую следующую задачу нужно выполнять. Когда задача выполнилась или перешла в режим ожидания, функция select) вызывается снова, и все повторяется.

Так, все процессорное время распределяется между разными задачами.

То есть несколько потоков выполнения мультиплексируются в одном потоке выполнения процессора. Никаких случайных перескакиваний и при этом - никаких простоев. Если есть хоть один готовый сокет, всегда будет что-то выполняться. Так, с помощью полностью синхронного кода мы добились реальной асинхронности в выполнении задач. Вот так синхронность перешла в асинхронность.

Многозадачность, реализованная через select(), называется совместной, или кооперативной (cooperative), потому что, прежде, чем начала выполняться следующая задача, нужно, чтобы предыдущая явно объявила о своей готовности отдать процессорное время другим задачам. То есть задачи очевидным образом согласуют между собой момент переключения задач. Еще более продвинутая версия многозадачности реализуется через Asincio также используя сопрограммы, о чем было описано в пункте 2.3.

## 2.5 Uploadfile и его применение

B FastAPI существует специальный компонент для обработки загрузки файлов, который называется UploadFile. Компонент UploadFile автоматически обрабатывает загрузку файла и предоставляет информацию о файле, такую как его имя.

UploadFile - это класс, предоставляемый FastAPI для работы с загружаемыми файлами. Он предоставляет удобный интерфейс для чтения и сохранения файлов, полученных от клиента.

Преимущества UploadFile:

Простота использования: UploadFile обеспечивает удобный интерфейс для работы с файлами. Он автоматически обрабатывает многие детали работы с файлами, такие как чтение и запись данных, управление памятью и т.д.

Интеграция с FastAPI: UploadFile тесно интегрирован с FastAPI и может быть легко использован вместе с функциями валидация запросов, обработка исключений и т.д.

Недостатки UploadFile:

Передача метаданных: Когда вы используете UploadFile, вместе с файлом передаются и его метаданные. Это может включать в себя такие вещи, как имя файла, тип содержимого и т.д. В некоторых случаях эта дополнительная информация может замедлить передачу данных.

Неэффективность при работе с большими файлами: UploadFile может быть неэффективным при работе с очень большими файлами. Это связано с тем, что FastAPI должен обрабатывать весь файл, прежде чем он сможет начать отправку ответа. Это может привести к задержкам и увеличению использования памяти.

Решение использовать байты вместо UploadFile для передачи данных основано на том, что передача данных в виде байтов может быть гораздо быстрее. Это особенно важно при работе с большими объемами данных, где любые задержки или неэффективности могут иметь значительное влияние на производительность вашего приложения. Кроме того, передача данных в виде байтов позволяет избежать передачи лишних метаданных, что также может ускорить процесс.

## 2.7 gRPC - Google Remote Procedure Call

gRPC - это новый и современный фреймворк для разработки масштабируемых, современных и быстрых АРІ и дословно переводится как система удаленного вызова процедур, разработанный компанией Google еще в 2015 году. Он позволяет программистам писать микросервисы на любом языке, который они хотят, сохраняя при этом возможность легко устанавливать связь между этими сервисами. И так же является реализацией удалённого вызова процедур (RPC), но уже в качестве транспорта использует новый НТТР/2. А для описания интерфейса используется Protocol Buffers (protobuf), где описывается структура для передачи, кодирования и обмена данных между представлениями описанными выше. Protocol Buffers проще, компактнее и быстрее, чем XML, поскольку осуществляется передача бинарных данных.

### 2.7.1 Использовать RPC и Protocol Buffers.

Первое преимущество использования Protocol Buffers заключается в гом, что они дают АР1 четко определенную и самодокументированную схему. Используя JSON, необходимо задокументировать содержащиеся в нем поля и их типы. Как и в случае с любой другой документацией есть риск, что она окажется неточной, неполной или устаревшей. Из API на Protobuf, можно сгенерировать код Python. То есть код никогда не будет рассинхронизирован с документацией.

Второе преимущество заключается в том, что при генерации кода Python из кода Protobuf, заодно получается некоторая базовую проверку. Например, сгенерированный код не принимает поля неправильного типа. Если для АРІ используется НТТР и JSON, то приходится писать код, который создает запрос, отправляет его, ожидает ответа, проверяет код состояния, а также анализирует и проверяет ответ. С помощью Protobuf можно генерировать код, который выглядит так же, как обычный вызов функции, но выполняет под капотом сетевой запрос.

Третье преимущество заключается в том, что платформа gRPC более эффективна, чем использование обычных НТТР-запросов. gRPC построен на основе НТТР/2, позволяющем выполнять несколько запросов параллельно поточно- безопасным способом. Сообщения RPC хранятся в бинарном виде и весят меньше, чем JSON. Кроме того, НТТР/2 имеет встроенное сжатие заголовков, а gRPC - встроенную поддержку потоковой передачи запросов и ответов.

Вероятно, самая важная причина, заключается в том, что можно определять свой АРI в терминах функций, а не НТТР-команд и ресурсов. В REST API необходимо решать, какие у есть ресурсы, как строить пути, как называть команды. Часто эти задачи могут быть решены множеством разных способов. Protobuf похож на JSON или XML в том, что все они являются способами форматирования данных. В отличие от JSON, Protocol Buffers имеют строгую схему и более компактны при передаче по сети.

B Protobuf для каждого типа данных, который требуется преобразовать в бинарный формат, создается соответствующее сообщение, где определяется структура данных и указывается имя и тип для каждого поля в этом сообщении. Затем, для описания логики взаимодействия между клиентом и сервером, определяется сервис и указываем методы, которые этот сервис предоставляет. Завершая этап подготовки протокола, компилируется описанные в протоколе классы. Просто введя команду с необходимой опцией для выбранного языка, компилятор автоматически генерирует код, освобождая разработчика от рутины ручного написания множества классов и методов.

### 2.7.2 Разновидности АРІ в gRPC

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

Unary RPC (Унитарное взаимодействие) - В унитарном RPC 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.

Картинка

Рисунок\_\_ - Схема унарной передачи данных.

Server Streaming RPC (Потоковое взаимодействие от сервера) – 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.

Картинка

Рисунок\_\_ Схема потоковой передачи данных от сервера.

Client Streaming RPC (Потоковое взаимодействие от клиента) - 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.

Картинка

Рисунок\_\_ - Схема потоковой передачи данных от клиента.

Bidirectional Streaming RPC (Двустороннее потоковое взаимодействие) -

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.

Картинка

Рисунок. 4 - Схема двунаправленной потоковой передачи данных.

Каждый из этих подходов предоставляет гибкость в выборе метода взаимодействия в зависимости от требований конкретной задачи. Кроме того, эти способы взаимодействия поддерживаются множеством языков программирования, что делает gRPC мощным инструментом для разработки распределенных систем с разнообразными потребностями.

## 2.6 Вывод по главе 2

2.6.1 В данной главе была представлена\_\_\_. Приведены качественные и количественные оценки работы\_\_\_. Стоит отметить, что \_\_\_, поэтому в следующей главе будет проведено исследование\_\_\_.

# ГЛАВА З РЕАЛИЗАЦИЯ ВЫШЕОПИСАННЫХ МЕТОДОВ ПЕРЕДАЧИ ДАННЫХ В КЛИЕНТ - СЕРВЕРНОЙ АРХИТЕКТУРЕ

В данной главе будут приведены исследования по различным технологиям реализации клиента (отправителя данных) и сервера (получателя и обработчика данных), для определения ключевых моментов влияющих на скорость передачи данных.

## 3.1 Реализация передачи данных по НТТР1

Текст

## 3.2 Реализация передачи данных по НТТР2

Текст

## 3.3 Сравнение НТТР1 и НТТР2

Текст

## 3.4 Вывод по главе 3

3.4.1 В данной главе были проведены исследования по различным технологиям реализации клиента и сервера для передачи больших данных. Определены ключевые моменты влияющие на скорость передачи данных.

# ГЛАВА 4 СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ РАЗЛИЧНЫХ МЕТОДОВ ПЕРЕДАЧИ ДАННЫХ

В данной главе\_\_\_.

## 4.1 Методология сравнения

Текст

## 4.2 Результаты передачи одного файла большого объема

Текст

## 4.3 Результаты передачи множества файлов

Текст

## 4.4 Выводы по главе 4

Текст

Недостатки UploadFile:

Передача метаданных: Когда вы используете UploadFile, вместе с файлом передаются и его метаданные. Это может включать в себя такие вещи, как имя файла, тип содержимого и т.д. В некоторых случаях эта дополнительная информация может замедлить передачу данных.

Неэффективность при работе с большими файлами: UploadFile может быть неэффективным при работе с очень большими файлами. Это связано с тем, что FastAPI должен обрабатывать весь файл, прежде чем он сможет начать отправку ответа. Это может привести к задержкам и увеличению использования памяти.

Решение использовать байты вместо UploadFile для передачи данных

основано на том, что передача данных в виде байтов может быть гораздо быстрее. Это особенно важно при работе с большими объемами данных, где любые задержки или неэффективности могут иметь значительное влияние на производительность вашего приложения. Кроме того, передача данных в виде байтов позволяет избежать передачи лишних метаданных, что также

может ускорить процесс.

2.6 Сервера ASGI - uvicorn и hypercorn

Текст

2.7 gRPC - Google Remote Procedure Call

gRPC - это новый и современный фреймворк для разработки масштабируемых, современных и быстрых АРІ и дословно переводится как система удаленного вызова процедур, разработанный компанией Google еще в 2015 году [5]. Он позволяет программистам писать микросервисы на любом языке, который они хотят, сохраняя при этом возможность легко устанавливать связь между этими сервисами. И так же является реализацией удалённого вызова процедур (RPC), но уже в качестве транспорта использует новый НТТР/2. А для описания интерфейса используется Protocol Buffers (protobuf), где описывается структура для передачи, кодирования и обмена

данных между представлениями описанными выше. Protocol Buffers проще, компактнее и быстрее, чем XML, поскольку осуществляется передача бинарных данных.

2.7.1 Использовать RPC и Protocol Buffers.

Первое преимущество использования Protocol Buffers заключается в гом, что они дают АР1 четко определенную и самодокументированную схему

[6]. Используя JSON, необходимо задокументировать содержащиеся в нем поля и их типы. Как и в случае с любой другой документацией есть риск, что она окажется неточной, неполной или устаревшей. Из API на Protobuf, можно сгенерировать код Python. То есть код никогда не будет рассинхронизирован с документацией.

Второе преимущество заключается в том, что при генерации кода

Python из кода Protobuf, заодно получается некоторая базовую проверку.

Например, сгенерированный код не принимает поля неправильного типа.

Если для АРІ используется НТТР и JSON, то приходится писать код, который

создает запрос, отправляет его, ожидает ответа, проверяет код состояния, а также анализирует и проверяет ответ. С помощью Protobuf можно генерировать код, который выглядит так же, как обычный вызов функции, но выполняет под капотом сетевой запрос.

Третье преимущество заключается в том, что платформа gRPC более эффективна, чем использование обычных НТТР-запросов. gRPC построен на основе НТТР/2, позволяющем выполнять несколько запросов параллельно поточно- безопасным способом. Сообщения RPC хранятся в бинарном виде и весят меньше, чем JSON. Кроме того, НТТР/2 имеет встроенное сжатие заголовков, а gRPC - встроенную поддержку потоковой передачи запросов и

ответов.

Вероятно, самая важная причина, заключается в том, что можно определять свой АР1 в терминах функций, а не НТТР-команд и ресурсов. В REST API необходимо решать, какие у есть ресурсы, как строить пути, как

называть команды. Часто эти задачи могут быть решены множеством разных способов. Protobuf похож на JSON или XML в том, что все они являются способами форматирования данных. В отличие от JSON, Protocol Buffers имеют строгую схему и более компактны при передаче по сети.

B Protobuf для каждого типа данных, который требуется преобразовать в бинарный формат, создается соответствующее сообщение, где определяется структура данных и указывается имя и тип для каждого поля в этом сообщении. Затем, для описания логики взаимодействия между клиентом и сервером, определяется сервис и указываем методы, которые этот сервис предоставляет. Завершая этап подготовки протокола, компилируется описанные в протоколе классы. Просто введя команду с необходимой опцией для выбранного языка, компилятор автоматически генерирует код, освобождая разработчика от рутины ручного написания множества классов и методов.

2.7.2 Разновидности АРІ в gRPC

gRPC предоставляет четыре основных способа взаимодействия между

клиентом и сервером, каждый из которых предназначен для различных сценариев использования.

Unary RPC (Унитарное взаимодействие) - В унитарном RPC 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.

Картинка

Рисунок. 1 - Схема унарной передачи данных.

Server Streaming RPC (Потоковое взаимодействие от сервера) - 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.

Рисунок

Рисунок. 2 - Схема потоковой передачи данных от сервера.

Client Streaming RPC (Потоковое взаимодействие от клиента) - 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.

Рисунок

Рисунок. 3 - Схема потоковой передачи данных от клиента.

Bidirectional Streaming RPC (Двустороннее потоковое взаимодействие) -

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.

Рисунок

Рисунок. 4 - Схема двунаправленной потоковой передачи данных.

Каждый из этих подходов предоставляет гибкость в выборе метода взаимодействия в зависимости от требований конкретной задачи. Кроме того, эти способы взаимодействия поддерживаются множеством языков программирования, что делает gRPC мощным инструментом для разработки распределенных систем с разнообразными потребностями.

2.6 Вывод по главе 2

2.6.1 В данной главе была представлена

Приведены качественные и количественные оценки работы

. Стоит отметить, что

, поэтому в следующей главе будет проведено исследование по подбору оптимальных параметров.

ГЛАВА З РЕАЛИЗАЦИЯ ВЫШЕОПИСАННЫХ

МЕТОДОВ ПЕРЕДАЧИ ДАННЫХ В КЛИЕНТ -

СЕРВЕРНОЙ АРХИТЕКТУРЕ

В данной главе будут приведены исследования по различным технологиям реализации клиента (отправителя данных) и сервера (получателя и обработчика данных), для определения ключевых моментов влияющих на скорость передачи данных.

3.1 Реализация передачи данных по НТТР1

Текст

3.2 Реализация передачи данных по НТТР2

Текст

3.3 Сравнение НТТР1 и НТТР2

Текст

3.4 Вывод по главе 3

3.4.1 В данной главе были проведены исследования по различным технологиям реализации клиента и сервера для передачи больших данных.

Определены ключевые моменты влияющие на скорость передачи данных.

ГЛАВА 4 СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ

РАЗЛИЧНЫХ МЕТОДОВ ПЕРЕДАЧИ ДАННЫХ

В данной главе

4.1 Методология сравнения Текст

4.2 Результаты передачи одного файла большого объема Текст

4.3 Результаты передачи множества файлов

4.4 Выводы по главе 4 Текст