Задача

Вы разрабатываете веб-сервис, который должен уметь три вещи:

- производить какие-то тяжелые вычисления
- работать с нейросетью
- принимать запросы от клиентов на эти самые действия

Тяжелые вычисления и работа с нейросетью — тяжелые синхронные операции, поэтому вы решили купить два (или больше) серверов и разместить этот тяжелый код на них. Вы не стали отделять кодовые базы — на каждой машине могут происходить как тяжелые вычислительные операции, так и работа с нейросетью, пусть это и не максимально эффективно. Давайте называть такой синхронный сервер *Consumer* — он принимает задачи, которых может быть несколько.

Отдельный сервер вы выделили под асинхронный сервер, который умеет принимать задачи от каких-то внешних клиентов, вызывать их на «синхронных» серверах, а затем возвращать результат. Будем называть его *Producer* — он посылает задачи и будет один.

RPC

Взаимодействие между Producer и Consumer можно настроить с помощью какого-нибудь высокоуровнего API — посылать команды по HTTP, парсить json и т.д. Но пока острой потребности в этом нет, и вы решили, что будете просто устанавливать TCP-соединение из Consumer в Producer и посылать команды набором байт.

Например, так:

```
b'calculator calculate 2 + 2'
```

На сервере эта команда будет разбиваться по пробелам и интерпретироваться примерно так:

```
calculator.calculate('2', '+', '2')
```

Первым аргументом в команде идет название класса, *calculator,* вторым — название его метода, *calculate* (@staticmethod или @classmethod), а все остальные — это *аргументы,* которые будут отправлены в метод.

Такой формат обмена данными называется RPC — Remote Procedure Call — мы *буквально* вызываем какой-то метод какого-то класса, который находится на удаленном сервере. Соответственно, результат работы этого метода calculator.calculate('2', '+', '2') должен быть преобразован в строку, затем переведен в байты, а затем записан в сокет, из которого пришел запрос. Затем сокет должен быть закрыт, чтобы не держать соединение. Это сделано для упрощения задачи.

Задачи

Кто-то внешний должен запрашивать выполнение задач у *Producer,* чтобы тот, в свою очередь, отправлял их в *Consumer* и ждал ответа. Обычно это какие-то запросы пользователей, но чтобы не поднимать свой асинхронный сервис, в этом задании мы сымитируем этот процесс:

```python
import queue
import random
from dataclasses import dataclass
from typing import Any

from faker import Faker

from tasks.common.event_loop import sleep

fake = Faker()
Faker.seed(0)
tasks_queue = queue.Queue()


@dataclass
class RpcTask:
    handler: str
    method: str
    value: Any


def client_payload():
    while True:
        # имитируем запросы пользователей, с примерной равной вероятностью добавляем ту или иную задачу
        # тут можно поиграться с вероятностью и значениями - разные задачи будут по разному долго исполняться
        if random.randint(0, 1) == 0:
            task = RpcTask(
                handler='fibonacci_generator',
                method='calculate',
                value=random.randint(100, 10000),
            )
        else:
            task = RpcTask(
                handler='neural_analyzer',
                method='polarity',
                value=fake.sentence(),
            )
        tasks_queue.put(task)
        print(f'В очередь была добавлена новая задача {task}')
        yield from sleep(0.5)  # можно покрутить время сна и посмотреть как поведет себя сервер
```

(этот код можно найти в tasks/rpc/producer/tasks_payload.py)

Разберем, что происходит в примере выше.

В нем есть RpcTask — это сущность, из которой очевидным образом можно сформировать RPC-команду, которую мы рассматривали выше. value в данном случае — это просто строка с разделенными пробелом аргументами.

Далее есть генератор client_payload(), который случайно ставит ту или иную задачу в очередь выполнения, а затем спит какое-то время. Все параметры распределения задач и время сна можно менять, чтобы посмотреть, как поведет себя сервер при разной нагрузке. Можно даже использовать какие-нибудь нелинейные функции в sleep().

Уже сейчас можно увидеть, какие задачи умеют выполнять наши Consumer: это вычисление какого-то по порядку числа Фибоначчи (вычислительная задача) и определение полярности, то есть позитивное оно или негативное, предложения (это условно задача на машинное обучение). Соответственно, наш client_payload() производит две эти задачи. Если с value для вычисления определенного числа Фибоначчи все понятно, то для генерации каких-то предложений для вычисления полярности пришлось использовать библиотеку Faker — она позволяет генерировать какие-то случайные выражения, не лишенные смысла. Например, мы можем сгенерировать имя, и это будет 'John' или 'Jack', а не 'asdbjbq@!s'. В нашем случае эта библиотека генерирует для нас предложения, которые похожи на настоящие, и поддаются анализу на полярность.

То есть у нас есть очередь задач, в которую асинхронно ставятся задачи раз в какое-то время.

А чтобы забирать задачи из очереди, есть RpcWorker — он умеет неблокирующе забирать задачи из очереди с помощью метода handle_tasks() и асинхронно ждать, если в текущий момент этих задач там нет. Он достает задачу, логирует ее и асинхронно вызывает метод call у RpcClient. Как только call выполнится, он залогирует результат выполнения задачи.

RpcClient — это класс, который с помощью метода call() умеет посылать команду заданному при его инициализации Consumer. При локальной разработке и тестировании Consumer находятся на одной машине, просто слушают разные порты. Поэтому для инициализации RpcClient нужно указать только порт. Далее call устанавливает соединение с выбранным Consumer, отправляет ему команду и считывает ответ. Все это, естественно, происходит асинхронно.

Consumer

Сервер, который блокирующе слушает заранее заданный сокет. То есть одновременно он может выполнять только одну команду. Его код уже написан за вас, потому что асинхронности в нем нет. Единственное, что вам нужно сделать в папке consumer/ — это дописать функцию setup_rpc_server. Это нужно, чтобы вы поняли, как можно сделать сервер, выполняющий команды по RPC.

Как выполнять задание

Это задание нужно делать также как и предыдущее — на заранее написаном нами Event Loop'е, без использования async/await и asyncio.

Запустить consumer можно такой командой:

```
./run_rpc_consumer.sh 9000
```

где 9000 — это порт, на котором consumer будет слушать.

Producer можно запустить такой командой:

```
./run_rpc_producer.sh 9000 9001
```

где 9000 и 9001 — это порты, на которых слушают Consumer.

Пример. Вы хотите сделать конфигурацию из 2 Consumer и одного Producer.

В одном терминале делаем:

```
./run_rpc_consumer.sh 9000
```

В другом:

```
./run_rpc_consumer.sh 9001
```

В третьем:

```
./run_rpc_producer.sh 9000 9001
```

Тесты на реализацию можно запустить этой командой:

```
./run_rpc_tests.sh
```