

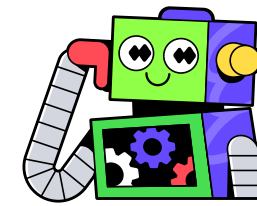
Правила взаимодействия

- 👉 Мы все равны
- 👉 Мы команда
- 👉 Эксперименты - это хорошо
- 👉 Много вопросов - это хорошо
- 👉 Готовиться к занятиям - это хорошо
- 👉 Все материалы - на платформе
- 👉 Постоянная обратная связь!

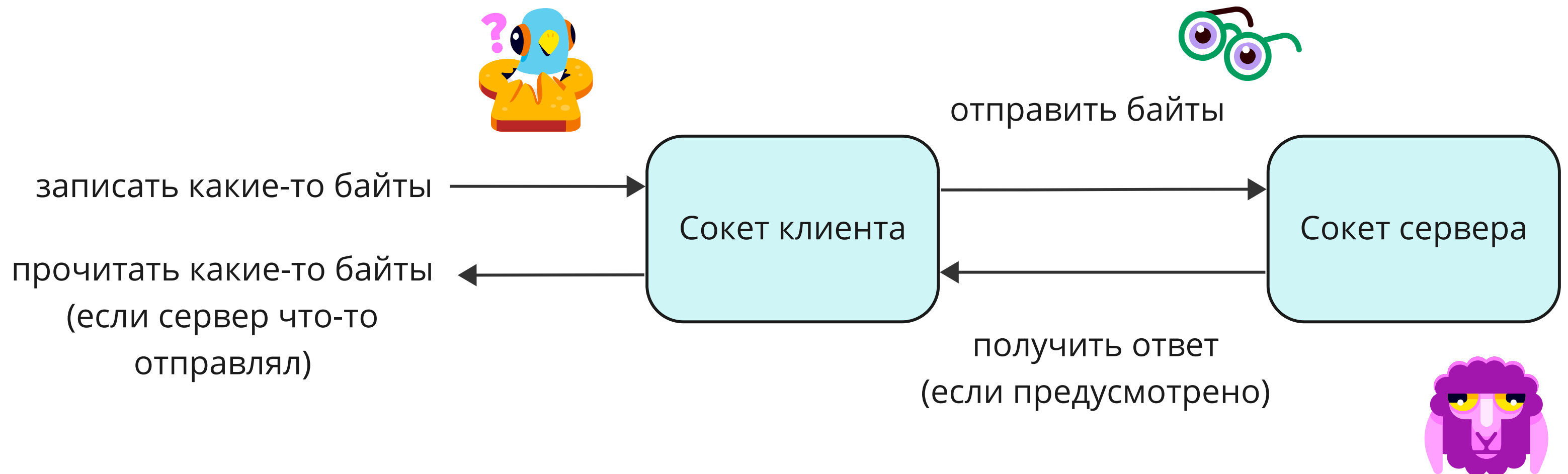
Что нас ждёт?

- 👉 Много кода
- 👉 Много разговоров о фундаментальном
- 👉 Обмен опытом
- 👉 Интерактив
- 👉 Экзамен

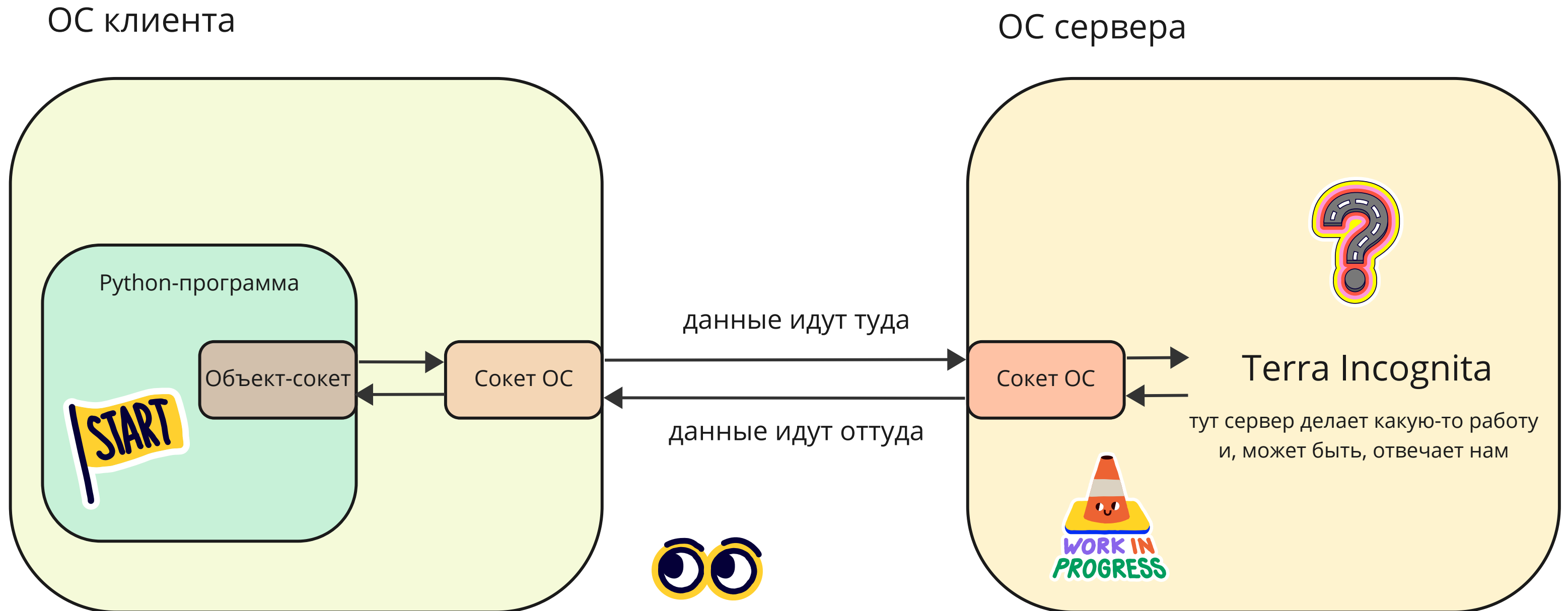
Сокеты: что это такое?



Это низкоуровневая абстракция, которая позволяет обмениваться данными по сети



Сокеты: клиент-серверное взаимодействие

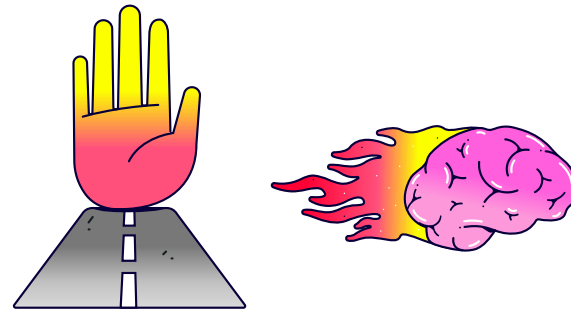


Важно понимать, что в Python сокеты - это питоновские объекты, которые реализуют интерфейс взаимодействия с сокетами ОС

Сокеты: какие бывают?



Сокеты в ОС бывают двух видов:



- ☞ блокирующие - не дают программе выполняться дальше до тех пор, пока взаимодействие с сокетом не будет завершено или пока не произойдет ошибка
- ☞ неблокирующие - позволяет программе выполняться дальше, чтобы потом вернуться к сокету и проверить результат выполнения операции

Для обоих видов сокетов в Python реализованы свои интерфейсы взаимодействия с ними



Сокеты: зачем 2 вида?



Почему нужны 2 вида сокетов? Дело в том, что они предоставляют разработчикам гибкость в выборе подходящего метода обработки сетевых операций в зависимости от требований и контекста приложения.



Блокирующие полезны когда:

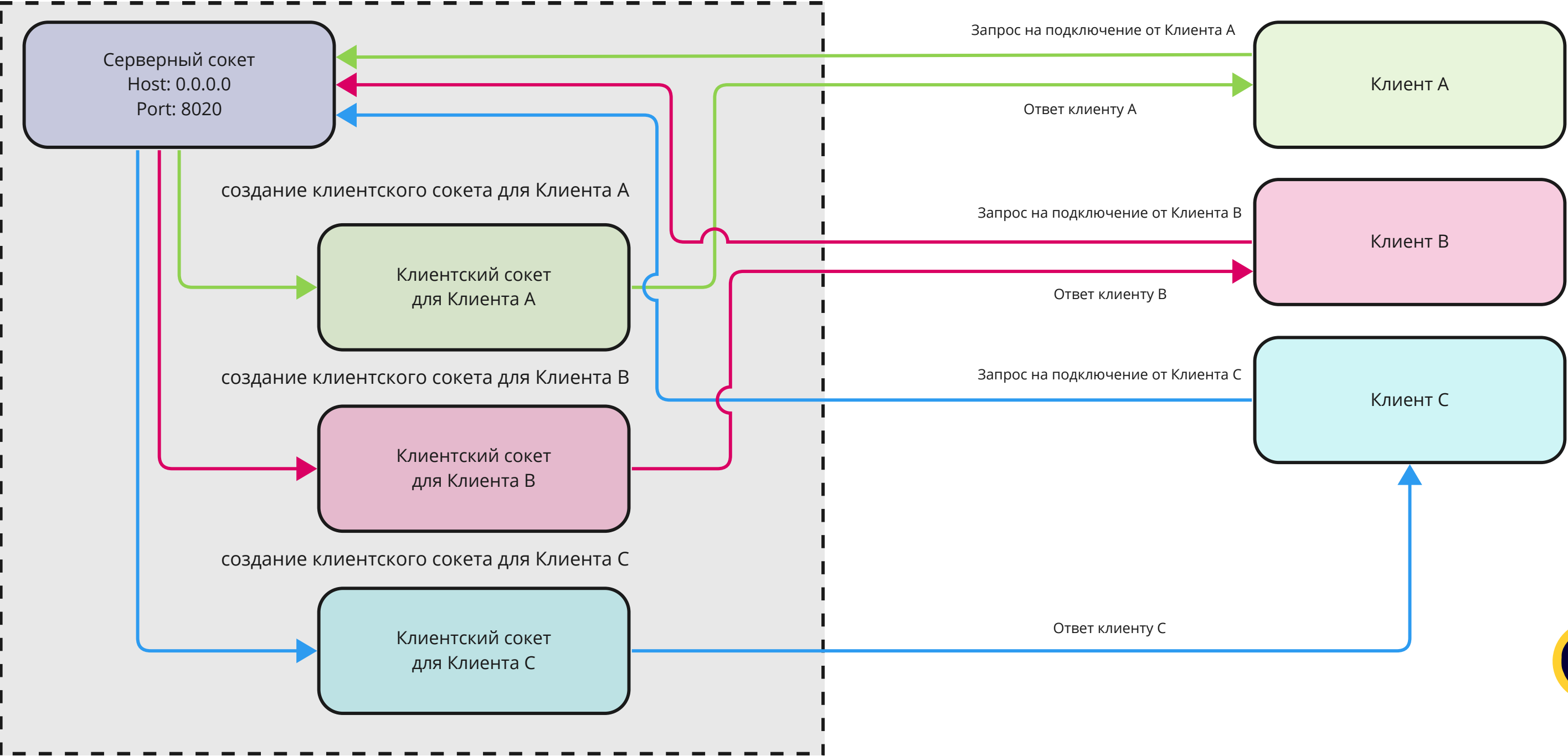
- ☞ нужно сделать попроще
- ☞ не ожидается высоких нагрузок
- ☞ ведется работа с потоками
- ☞ важна строгая последовательность

Неблокирующие нужны когда:

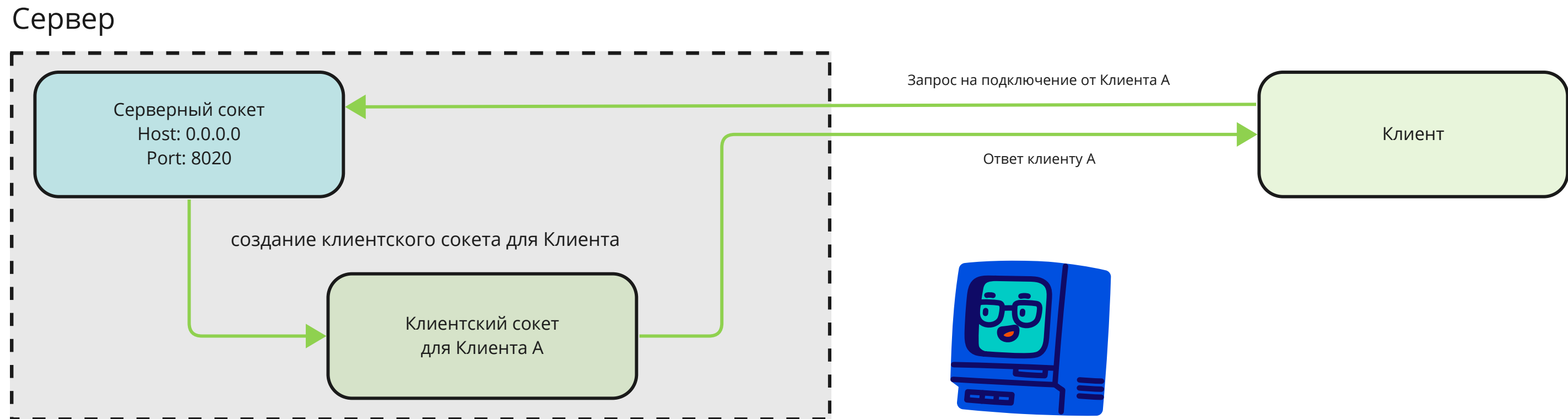
- ☞ нужна производительность и масштабируемость
- ☞ ожидаются высокие нагрузки
- ☞ нет требований по строгой последовательности

Сокеты: клиентские и серверный

Сервер



Сокеты: клиентские и серверный



Задача серверного сокета - слушать входящие запросы на соединение и создавать отдельный сокет (клиентский) для взаимодействия с клиентом.

Задача клиентского сокета - передавать данные между клиентом и сервером.

Простой пример программы (сервер)

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # address type (host + port), protocol TCP
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # reuse port after stopping program
server_socket.bind(("127.0.0.1", 8020))
print("Start listening...")
server_socket.listen() # start listening incoming connections - BLOCK PROGRAM!
print("Listening...")
client_socket, client_address = server_socket.accept() # client socket and address data
print("Something happened!")
print(f"New connection from: {client_address}")
```

☞ **socket.AF_INET** - указывает, что сокет будет использовать IPv4 (Internet Protocol v4). Это означает, что адресация в сети будет осуществляться через IP-адреса, которые состоят из четырех чисел, разделенных точками (например, 127.0.0.1)

☞ **socket.SOCK_STREAM** - обозначает тип сокета, который поддерживает последовательные, надежные и двусторонние потоковые передачи данных. Этот параметр связан с использованием протокола TCP, который гарантирует, что данные будут доставлены без ошибок и в правильном порядке.



Простой пример программы (сервер)

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # address type (host + port), protocol TCP
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # reuse port after stopping program
server_socket.bind(("127.0.0.1", 8020))
print("Start listening...")
server_socket.listen() # start listening incoming connections - BLOCK PROGRAM!
print("Listening...")
client_socket, client_address = server_socket.accept() # client socket and address data
print("Something happened!")
print(f"New connection from: {client_address}")
```

Опции сокета - более тонкие и низкоуровневые параметры настройки сокета на уровне ОС.

☞ **socket.SOL_SOCKET** — это уровень (или область) опции сокета. В данном случае, он указывает, что опция применяется непосредственно к API сокетов, а не к какому-то конкретному протоколу, как TCP или UDP.

☞ **socket.SO_REUSEADDR** — это конкретная опция, которую вы устанавливаете для сокета. Эта опция позволяет повторно использовать локальные адреса. В контексте сервера это означает, что вы можете перезапустить сервер и повторно привязаться к тому же порту. Без SO_REUSEADDR, если вы попытаетесь перезапустить сервер сразу после его остановки, вы можете получить ошибку "address already in use"


☞ **1** - означает что мы переводим опцию в состояние "ВКЛ"




Простой пример программы (сервер)


```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # address type (host + port), protocol TCP
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # reuse port after stopping program
server_socket.bind(("127.0.0.1", 8020))
print("Start listening...")
server_socket.listen() # start listening incoming connections - BLOCK PROGRAM!
print("Listening...")
client_socket, client_address = server_socket.accept() # client socket and address data
print("Something happened!")
print(f"New connection from: {client_address}")
```

 **server_socket.bind(("127.0.0.1", 8020))** - привязка сокета к конкретному IP-адресу и порту на машине, на которой выполняется код. Это означает, что сокет будет слушать входящие соединения на IP-адресе 127.0.0.1 (localhost) и порту 8020.

Вы можете задать любой IP-адрес, который присвоен одному из сетевых интерфейсов вашего сервера. Например:

 **0.0.0.0:** Привязка сокета к этому адресу означает, что сервер будет принимать все входящие соединения на любом IP-адресе, который присвоен машине. Это позволяет серверу быть доступным с любого другого компьютера в сети или интернете (при условии, что нет фаерволов или других сетевых ограничений).


 **Конкретный IP-адрес:** Если у вашего сервера несколько сетевых интерфейсов (например, один смотрит в интернет, а другой — во внутреннюю сеть), вы можете выбрать конкретный IP-адрес одного из этих интерфейсов для привязки сокета.



Простой пример программы (сервер)

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # address type (host + port), protocol TCP
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # reuse port after stopping program
server_socket.bind(("127.0.0.1", 8020))
print("Start listening...")
server_socket.listen() # start listening incoming connections - BLOCK PROGRAM!
print("Listening...")
client_socket, client_address = server_socket.accept() # client socket and address data
print("Something happened!")
print(f"New connection from: {client_address}")
```

 **server_socket.listen()** - сообщаем операционной системе, что данный сокет должен быть использован для прослушивания входящих соединений.

Вот что происходит:

1. **Прослушивание порта:** Сокет начинает "слушать" на порту, к которому он был привязан с помощью метода `bind()`. Это означает, что он ожидает входящие сетевые запросы на этот порт.
2. **Очередь входящих соединений:** Метод `listen()` опционально принимает один аргумент — размер "бэклога" (backlog). Этот параметр определяет максимальное количество входящих соединений, которые могут быть помещены в очередь ожидания, если сервер временно не может их обработать.



Простой пример программы (сервер)

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # address type (host + port), protocol TCP
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # reuse port after stopping program
server_socket.bind(("127.0.0.1", 8020))
print("Start listening...")
server_socket.listen() # start listening incoming connections - BLOCK PROGRAM!
print("Listening...")
client_socket, client_address = server_socket.accept() # client socket and address data
print("Something happened!")
print(f"New connection from: {client_address}")
```

Метод `accept()` используется на серверном сокете для принятия входящего соединения от клиента. Он блокирует выполнение программы до тех пор, пока не поступит входящее соединение от клиента. В момент установления соединения метод `accept()`:

- 👉 Извлекает соединение из очереди ожидающих соединений
- 👉 Создает новый клиентский сокет для извлеченного из очереди соединения
- 👉 Возвращает кортеж с адресом и портом клиента для последующего использования (например логирования)



Простой пример программы (клиент)

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('127.0.0.1', 8020))
s.sendall("ping".encode("utf-8"))
s.close()
```

- 👉 Устанавливаем соединение с сервером по адресу 127.0.0.1 и портом 8020
- 👉 Отправляем в виде байтов строку ping
- 👉 Закрываем сокет, чтобы не блокировать сервер



Какие проблемы сейчас есть у сервера и клиента?

Сервер:

- ☞ Принимает соединение и сразу прекращает работу
- ☞ Ничего не отдает на сторону клиента
- ☞ По факту работает только с одним соединением

Клиент:

- ☞ Отправив данные на сторону сервера сразу прекращает работу
- ☞ В случае получения ответа от сервера никак его не обрабатывает

Что можно сделать?

Сервер:

- 👉 Вычитывать данные из сокета
- 👉 Что-то отвечать клиенту
- 👉 Обработав одного клиента брать в работу следующего

Клиент:

- 👉 Реализовать возможность отправлять больше данных на сервер
- 👉 Обработывать ответ сервера

Рекомендованная домашка



1

Напишите клиент таким образом, чтобы можно было отправлять через него на сервер данные до тех пор, пока пользователем не будет введено слово STOP.



2

Одновременно запустите несколько клиентов в разных процессах и отправляйте из каждого из них сообщения на сервер. Что вы наблюдаете? Как вы думаете, почему так происходит?

Ещё рекомендованная домашка

3

Напишите клиент и сервер для передачи содержимого текстовых файлов.

Схема работы клиента:

- Запросить у пользователя путь к файлу, который будем передавать
- Запросить у сервера имена текстовых файлов, которые уже существуют на нем
- Запросить у пользователя имя файла в который будем писать текстовый файл
- Прочитать данные из текстового файла и передать их на сервер

Схема работы сервера:

- По команде клиента отдать список существующих текстовых файлов на сервере
- Принять имя файла от клиента, в который будет осуществляться запись данных
- Если файла не существует - создать его, если существует - уточнить, хочет ли клиент переписать файл или дописать данные в существующий
- Принять данные и записать их в файл

Сервер и клиент должны быть устойчивы к возникновению ошибок на любой стороне: обрыв соединения,
внезапное закрытие сокета и т.д.

