

Программирование сокетов

Сокеты

Для работы с сетью на низком уровне используются сокеты (socket). Сокет - это название программного интерфейса (API) для обеспечения обмена данными между процессами. Процессы при этом могут выполняться как на одном хосте, так и на различных.

Хорошей аналогией сокета является файловый дескриптор - некоторая абстракция, для работы с файлами в ОС. Вы можете открыть или закрыть файловый дескриптор, можете прочитать или записать туда данные и т.д.

Использование сокета у клиента немного отличается от сервера. Во время разработки сервера вы увидите эту разницу.

NOTE

Подробности по использованию сокетов в Python можно найти [в официальной документации](#).

TCP-клиент

Давайте напишем первого сетевого клиента, который будет подключаться к серверу miminet.ru на порт 80 (TCP), отправлять HTTP-запрос, получать HTTP-ответ и печатать содержимое ответа в консоль. Схема работы показана на рисунке ниже.

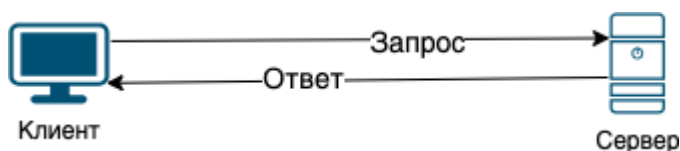


Figure 1. Схема работы TCP-клиента.

NOTE

HTTP (HyperText Transfer Protocol) - протокол для коммуникации с веб-сервером. HTTP-запрос - запрос отправляемый клиентом на веб-сервер. В ответ на такой запрос веб-сервер отправляет данные (HTML страницу, JS код, картинки, видео и т.д.) или отправляет информационное сообщение.

NOTE

По умолчанию, веб-сервер для работы использует порт 80 (TCP).

Нам необходимо выполнить следующие действия:

1. Создать TCP сокет.
2. Подключиться к веб-серверу miminet.ru на порт 80.
3. Отправить HTTP-запрос.
4. Получить ответ и вывести его на экран.

```
#!/usr/bin/python
```

```
import socket

HOST = "miminet.ru"
PORT = 80                                # HTTP port

ip_addr = socket.gethostbyname(HOST)     # Get IP address by name
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Create TCP socket
s.connect((ip_addr, PORT))               # Connect to miminet.ru
s.send(b'GET / HTTP/1.0\n\n')           # Send HTTP-request
data = s.recv(4096)                     # Recieve data from miminet.ru
print(data)                             # Print data to console
```

Вы можете скопировать этот код и попробовать выполнить его у себя. В результате его работы в консоль должно вывестись следующее сообщение:

```
(venv) ScrumBook:src ilya2$ python tcp-client-1.py
b'HTTP/1.1 301 Moved Permanently\r\nServer: nginx/1.25.4\r\nDate: Wed, 16 Oct 2024
08:40:37 GMT\r\nContent-Type: text/html\r\nContent-Length: 169\r\nConnection:
close\r\nLocation: https://miminet.ru/\r\n\r\n<html>\r\n<head><title>301 Moved
Permanently</title></head>\r\n<body>\r\n<center><h1>301 Moved
Permanently</h1></center>\r\n<hr><center>nginx/1.25.4</center>\r\n</body>\r\n</html>\r
\n'
```

Я постарался прокомментировать каждую строку, но давайте подробнее разберем, что мы делаем и что происходит

```
import socket
```

Импортируем модуль `socket`. Как раз он нам позволит работать с сокетами.

```
ip_addr = socket.gethostbyname(HOST)
```

Для создания TCP соединения необходимо знать IP-адрес хоста и порт, на который мы хотим подключиться. Пока мы знаем только имя хоста `miminet.ru` и порт `80`. Функция `gethostbyname` позволяет по имени определить IP-адрес хоста. Т.е. по имени `miminet.ru` она вернет IP-адрес хоста. Сохраним его в переменную `ip_addr`.

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Данный код создает TCP сокет и сохраняет его в переменную `s`. Функция `socket` позволяет создавать не только TCP или UDP сокеты. Она принимает два обязательных параметра:

- `socket.AF_INET` - когда мы хотим открыть сокет из семейства протоколов TCP/IP
- `socket.SOCK_STREAM` - для создания TCP-сокета.

И так, сокет создан, IP-адрес и порт известны, можем подключаться.

```
s.connect((ip_addr, PORT))
```

Функция `connect` устанавливает TCP соединения (3-х разовое рукопожатие SYN, SYN+ACK, ACK) на заданный IP-адрес и порт. Обратите внимание, что функция `connect` принимает только один порт - это порт назначения. Т.е. порт, на который мы устанавливаем соединение. У клиентов (тот кто инициирует установку соединения) порт источника выбирается случайным образом.

```
s.send(b'GET / HTTP/1.0\n\n')
```

Функция `send` отправляет данные в установленное TCP соединение. В данном случае мы отправили HTTP-запрос на веб-сервер `miminet.ru`. HTTP - это текстовый протокол, поэтому, запрос можно написать прямым текстом. Два подряд идущие перевода строки `\n\n` означают окончание HTTP-запроса. В этом HTTP-запросе мы хотим получить главную страницу `miminet.ru`.

```
data = s.recv(4096)
```

Получаем данные из сокета. Функция `recv` по умолчанию работает в блокирующем режиме. Это означает, что вызвав эту функцию программа будет ожидать, пока не придут данные. Так как заранее не известно, сколько данных придет, указал 4096 байт. Если данных придет меньше, вернут их, если придет больше, вернут не более 4096 байт. Полученные данные сохраняем в переменную `data`.

```
print(data)
```

Печатаем полученные данные в консоль.

NOTE

На самом деле можно опустить вызов функции `gethostbyname`, так как функция `connect` может самостоятельно по имени определить IP адрес. Я решил оставить `gethostbyname` для наглядности.

Обработка ошибок

Код из примера выше годится только для примера. Не трудно заметить, что там нет обработки ошибок. Давайте добавим обработку ошибок и посмотрим, что может пойти не так.

```
#!/usr/bin/python

import socket
```

```
HOST = "miminet.ru"
PORT = 80

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    s.connect((HOST, PORT))
    s.send(b'GET / HTTP/1.0\n\n')
    data = s.recv(4096)
    print(data)
except Exception as e:
    print ("Socket error: " + str(e))
```

Ошибка преобразования имени в IP адрес

Попробуем вместо имени хоста miminet.ru подставить несуществующее имя, например, miminet.rus.

```
HOST = "miminet.rus"
```

Для этого изменим имя хоста в переменной HOST и запустим наш код. В результате мы получим ошибку:

```
(venv) ScrumBook:src ilya2$ python tcp-client-2-hostname.py
Socket error: [Errno 8] nodename nor servname provided, or not known
```

Она означает, что нам не удалось по имени хоста определить IP-адрес. Дальнейшая установка соединения бессмысленно, так как мы не знаем IP-адрес хоста.

NOTE

Для воспроизведения следующих ошибок верните имя в переменной HOST обратно на miminet.ru

Ошибка подключения

Для воспроизведения ошибки подключения поменяем переменную PORT с 80 на 81 (можно и 81 и многие другие порты, которые закрыты). Запустим наш код и увидим ошибку подключения:

```
(venv) ScrumBook:src ilya2$ python tcp-client-2-port81.py
Socket error: [Errno 61] Connection refused
```

Данная ошибка сообщает, что на удаленной стороне нет программы, которая готова работать на указанном порту. В моем случае - это порт 81. Нет смысла продолжать выполнять программу и пытаться отправить данные. Ошибка появляется, когда во время

установки TCP соединения клиент получает пакет с флагом RST. Из хорошего, данная ошибка появляется быстро. Т.е. наш хост отправил SYN пакет, в ответ получил RST и сообщил об этом нам.

А теперь попробуем установить TCP соединение на порт, который не будет отвечать пакетом с флагом RST. Посмотрим, как себя поведет наша программа. Для этого на сервере `miminet.ru` настроен фаервол, который отбрасывает все входящие TCP пакеты с портом назначения равным 8000.

Поменяем переменную `PORT` на 8000 и запустим нашу программу. После длительного ожидания появляется ошибка:

```
(venv) ScrumBook:src ilya2$ python tcp-client-2-port8000-1.py
Socket error: [Errno 60] Operation timed out
```

Ошибка означает, что не удалось установить TCP соединение. Такую ошибку можно наблюдать, когда пакеты по какой-то причине не доходят до сервера. Либо они блокируются фаерволом, либо сервер просто выключен.

Это неприятная ошибка! Сокет и все его функции, включая `connect`, по умолчанию работают в блокирующем режиме. Это когда программа вызывает функцию и ждет, пока эта функция не завершит свою работу. И когда пакеты во время установки соединения вот так теряются, то вся программа зависает. В моем случае программа зависла на 75 секунд.

```
(venv) ScrumBook:src ilya2$ time python tcp-client-2-port8000-1.py
Socket error: [Errno 60] Operation timed out
```

```
real    1m15.779s
user    0m0.028s
sys     0m0.012s
```

NOTE

Если у вас Linux или MacOS, то для замера времени исполнения программы, перед запуском напишите `time`. Общее время исполнения программы будет отображаться в строке `real`.

Чтобы решить проблему с подвисанием, можно запустить работу с сокетом в отдельном потоке. Но, даже в отдельном потоке установка соединения может происходить аж 70 секунд. Это очень долго! Современные сети работают на много быстрее, чтобы ждать столько времени перед тем, как будет принято решение о невозможности установить соединение. Для уменьшения таймаута воспользуемся функцией `settimeout`.

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(5)
```

Сразу после создания сокета установим таймаут на блокирующие операции в 5 секунд. Теперь функция `connect` ожидает всего 5 секунд, после чего сообщает об ошибке и завершает

свою работу.

```
(venv) ScrumBook:src ilya2$ time python tcp-client-2-port8000-2.py
Socket error: timed out
```

```
real    0m5.055s
user    0m0.032s
sys     0m0.013s
```

NOTE

Установка таймаута в 0 переведет сокет в неблокирующий режим. В этом случае нужно будет поменять схему работы с сокетом. Об этом будет подробнее рассказано дальше по курсу.

При работе с сокетом я всегда рекомендую уменьшать таймауты до приемлемого значения. Современные компьютерные сети позволяют на много быстрее определить невозможность установки соединения и сообщить об этом.

Обработка отправки данных (send)

Функция `send` обычно отработывает без сбоя. Но вот что стоит о ней знать! Когда вы вызываете `send`, то данные не передаются приложению на другом конце сокета. Функция `send` только помещает данные в буфер для отправки. И все.

После того как функция `send` поместила данные на отправку, соединение может быть уже разорвано и, соответственно, никакие данные никуда не будут переданы. Учтите этот момент!

Обработка получения данных (recv)

Функция `recv` принимает один обязательный аргумент - это максимальное количество байт, которое можно вернуть. В случае ошибки функция `recv` вернёт 0 байт данных. Это будет означать, что соединение было закрыто и от туда больше ничего не может быть получено.

Особо внимание стоит обратить на работу функции `recv`. Функция `recv` - блокирующая функция и она будет ожидать данные вечно (либо пока соединение не будет закрыто). Теоретически, вызвав `recv` вы можете вечно ожидать, пока она что-то вернет.

Чтобы воспроизвести проблему с `recv` изменим наш код на следующий:

```
#!/usr/bin/python

import socket

HOST = "miminet.ru"
PORT = 80

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(5)
```

```
try:
    s.connect((HOST, PORT))
    data = s.recv(4096)
    print(data)
except Exception as e:
    print ("Socket error: " + str(e))
```

В этом коде мы убрали отправку HTTP-запроса (`s.send`) и сразу ожидаем данные. Запустите этот код. Программа будет ожидать данные до тех пор, пока сокет не будет закрыт или пока не истечет таймаут.

```
(venv) ScrumBook:src ilya2$ time python tcp-client-2-recv.py
Socket error: timed out

real    0m5.079s
user    0m0.028s
sys     0m0.011s
```

В этом случае ожидание длилось 5 секунд - время установленного таймаута.

ТСР-клиент с обработкой ошибок

Давайте посмотрим, как будет выглядеть наш ТСР-клиент с обработкой указанных ошибок:

```
#!/usr/bin/python

import socket

HOST = "miminet.ru"
PORT = 80

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(5)

try:
    s.connect((HOST, PORT))
    s.send(b'GET / HTTP/1.0\n\n')
    data = s.recv(4096)

    if not data:
        raise RuntimeError("socket connection broken")

    print(data)
except Exception as e:
    print ("Socket error: "+str(e))
```

```
s.settimeout(5)
```

Чтобы не ждать слишком долго блокирующих операций (при невозможности установить соединение и когда нечего читать в буфере приема).

```
if not data:  
    raise RuntimeError("socket connection broken")
```

Проверяем результат работы функции `recv` и в случае ошибки сообщаем, что соединение было закрыто.

В целом, уже не плохо!

Правда, проблема с `recv` до конца не решена. Если данные не поступят, то `recv` будет их ждать 5 секунд.

Проверка доступности данных (select)

Можно работу с сокетами выделить в отдельный поток (`thread`) и не переживать о блокировке кода. Но, это перенос проблемы из одного места в другое. Для решения данной проблемы используется функция `select`.

Функция `select` позволяет проверить наличия данных в буфере, что дает возможность вызывать `recv` только тогда, когда буфер не пуст и избегать ненужных ожиданий. Ниже представлен код с использованием `select`

```
#!/usr/bin/python  
  
import socket  
import select  
  
HOST = "miminet.ru"  
PORT = 80  
  
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  
s.settimeout(5)  
  
try:  
    s.connect((HOST, PORT))  
    s.send(b'GET / HTTP/1.0\n\n')  
  
    rdy = select.select([s], [], [], 2)  
    if not rdy[0]:  
        raise RuntimeError("no response")  
  
    data = s.recv(4096)
```



```
if not data:
    raise RuntimeError("socket connection broken")

print(data)

except Exception as e:
    print ("Socket error: "+str(e))
```

Функция select принимает 4 аргумент:

- список дескрипторов, готовых для чтения
- список дескрипторов, готовых для записи
- список дескрипторов которые в исключительном состоянии (exceptional condition)
- время ожидания (float)

```
rdy = select.select([s], [], [], 2)
if not rdy[0]:
    raise RuntimeError("no response")
```

Как и многие другие функции, `select` - блокирующая функция. Код выше означает - жать 2 секунды или пока в сожете `s` не появятся данные для чтения.

Проверка нужна для того, чтобы определить, функция `select` завершилась по таймауту (2 секунды) или появились данные для чтения.

NOTE

функция `select` работает с дескрипторами и ей все равно, это сокет, файловый дескриптор или дескриптор для ввода/вывода с консоли.

Таким образом, у нас получился следующий TCP-клиент:

- если все хорошо, то все хорошо
- если невозможно установить TCP соединение, сразу сообщаем об этом
- если TCP соединение не устанавливается 5 секунд (вместо 70), прекращаем работу с ошибкой
- если данные не приходят в ответ на запрос, ждем 2 секунды (вместо 5) и прекращаем работу.

Это уже на много лучше того, что было изначально.

NOTE

Вся сила функции `select` совсем не в том, что мы ждем всего 2 секунды. Она раскрывается при работе с множеством сокетов. Что бы для каждого сокета не выделять отдельный поток, используя `select` всю работу можно организовать в одном потоке.

UDP-клиент

UDP-клиент во многом очень похож на TCP-клиент. Напишем программу, которая узнает точное время от одного из серверов времени. Для работы с сервером времени используется UDP протокол.

NOTE

Для синхронизации времени ОС используют SNTP (Simple Network Time Protocol) протокол. Сервер SNTP использует для работы порт 123 (UDP). SNTP работает по схеме запрос-ответ. Клиент отправляет запрос на сервер, а в ответ получает информацию о точном времени.

```
import socket
import struct
import time
import select

NTP_SERVER = "2.ru.pool.ntp.org"
PORT = 123
TIME1970 = 2208988800

client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

data = '\x1b' + 47 * '\0'

try:
    client.sendto( data.encode('utf-8'), (NTP_SERVER, PORT))
    rdy = select.select([client], [], [], 0.9)

    if not rdy[0]:
        raise RuntimeError("socket recv broken")

    data, address = client.recvfrom(1024)

    if data:
        print ('Response received from:', address)
        t = struct.unpack( '!12I', data )[8]
        t -= TIME1970
        print ('\tTime=%s' % time.ctime(t))
except Exception as e:
    print ("Socket error: " + str(e))
```

Необходимо выполнить следующие действия:

1. Создать UDP сокет.
2. Отправить запрос на SNTP сервер.
3. Получить ответ, достать полученное время и вывести его на экран.

Обратите внимание, в отличие от TCP-клиента у UDP-клиента нет необходимости устанавливать соединение. Как мы знаем, UDP протокол не поддерживает их.

```
NTP_SERVER = "2.ru.pool.ntp.org"  
PORT = 123  
TIME1970 = 2208988800
```

Переменные NTP_SERVER и PORT содержат имя сервера и порт, на который мы будем отправлять SNTP запрос. Переменная TIME1970 содержит количество секунд прошедших с 1 Января 1900 года по 1 Января 1970 года.

```
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Создание UDP-сокета. Когда мы создавали TCP-сокет, то вторым параметром указывали socket.SOCK_STREAM, для создания UDP-сокета нужно указать socket.SOCK_DGRAM.

```
data = '\x1b' + 47 * '\0'
```

Запрос, который мы будем отправлять на SNTP сервер. Не суть, что именно означает такой запрос. Пока главное понять, что SNTP сервер получив такой запрос сформирует и отправит SNTP-ответ, в котором будет указано точное время.

NOTE | Более подробно про формат SNTP-пакета можно почитать в [RFC 1769](#).

```
client.sendto( data.encode('utf-8'), (NTP_SERVER, PORT))
```

Функция `sendto` используется для отправки данных по UDP. Она принимает второй аргумент аналогичный тому, который принимает функция `connect` при установке TCP-соединения - это имя сервера или его IP-адрес и порт назначения. Еще раз обратите внимание, перед отправкой данных по UDP соединение не устанавливается.

Отсутствие установки соединения приводит к тому, что после создания UDP-сокета нельзя вызывать функцию `recv` или её аналог. На какой входящий порт ожидать UDP-пакет? А когда мы вызовем функцию `sendto`, то ОС отправит пакет на заданный IP-адрес и порт, а порт источника выберет случайным образом. И именно после этого момента можно будет вызывать функции для получения данных. Теперь, если придет UDP пакет на наш случайно выбранный порт и IP-адрес и порт источника, при этом, будут идентичны тем, что мы указали при `sendto` - то ОС передаст нам пакет на обработку.

```
rdy = select.select([client], [], [], 0.9)  
  
if not rdy[0]:  
    raise RuntimeError("socket recv broken")
```

UDP - ненадежный протокол передачи данных. Отправив запрос на SNTP сервер не факт, что он дойдет. И еще нет уверенности в том, что ответ не потеряется. Поэтому, есть не малая вероятность вызывать функцию чтения из сокета и зависнуть там на долго. Мы уже знакомы с `select`, поэтому воспользуемся этой функцией для проверки доступности данных в буфере на чтение.

Я намеренно установил время ожидания менее 1 секунды, чтобы показать такую возможность. Некоторые разработчик, когда им нужно подождать, например, 0.5 секунды, пишут `select([], [], [], 0.5)`.

```
data, address = client.recvfrom(1024)
```

Функция `recvfrom` аналогична `recv`, только еще возвращает пару IP-адрес и порт источника.

```
if data:
    print ('Response received from:', address)
    t = struct.unpack( '!12I', data)[8]
    t -= TIME1970
    print ('\tTime=%s' % time.ctime(t))
```

Проверяем, есть ли данные в полученном пакете. Если есть:

- печатаем IP-адрес и порт источника
- достаем из SNTP пакета время с сервера (кому интересно, смотрите подробности в [RFC 1769](#))
- вычитаем из времени 70 лет. SNTP сервер считает время в секундах от 1 Января 1900 года, а модуль `time` ожидает, что на вход поступит количество секунд прошедших с 1 Января 1970 года. Поэтому нужно из времени от SNTP сервера вычесть 70 лет.
- печатаем время в консоль.

Результат работы программы представлен ниже:

```
(venv) ScrumBook:src ilya2$ python udp-client-1.py
Response received from: ('192.36.143.130', 123)
Time=Wed Oct 16 17:50:43 2024
```

NOTE

Если у вас в нашем примере очень часто не приходят пакеты от сервера времени, попробуйте поменять его на `3.ru.pool.ntp.org` или `1.ru.pool.ntp.org`.

Отправка на несколько хостов

Когда мы вызываем функцию `recvfrom`, то помимо самих данных приходит информация об отправителе. В частности, IP-адрес и порт отправителя.

Может показаться - зачем, ведь мы помним, кому мы отправили данные? Дело в том, что UDP-сокеты позволяют отправлять данные разным клиентам и на разные порты. Давайте перепишем наш UDP-клиент таким образом, чтобы он отправлял данные сразу на 3 разных сервера времени:

- 1.ru.pool.ntp.org
- 2.ru.pool.ntp.org
- 3.ru.pool.ntp.org

```
import socket
import struct
import time
import select

NTP_SERVER1 = "1.ru.pool.ntp.org"
NTP_SERVER2 = "2.ru.pool.ntp.org"
NTP_SERVER3 = "3.ru.pool.ntp.org"
PORT = 123
TIME1970 = 2208988800

client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

data = '\x1b' + 47 * '\0'

try:
    client.sendto( data.encode('utf-8'), (NTP_SERVER1, PORT))
    client.sendto( data.encode('utf-8'), (NTP_SERVER2, PORT))
    client.sendto( data.encode('utf-8'), (NTP_SERVER3, PORT))

    for i in range(3):
        rdy = select.select([client], [], [], 0.9)

        if not rdy[0]:
            raise RuntimeError("socket recv broken")

        data, address = client.recvfrom(1024)

        if data:
            print ('Response received from:', address)
            t = struct.unpack( '!12I', data)[8]
            t -= TIME1970
            print ('\tTime=%s' % time.ctime(t))

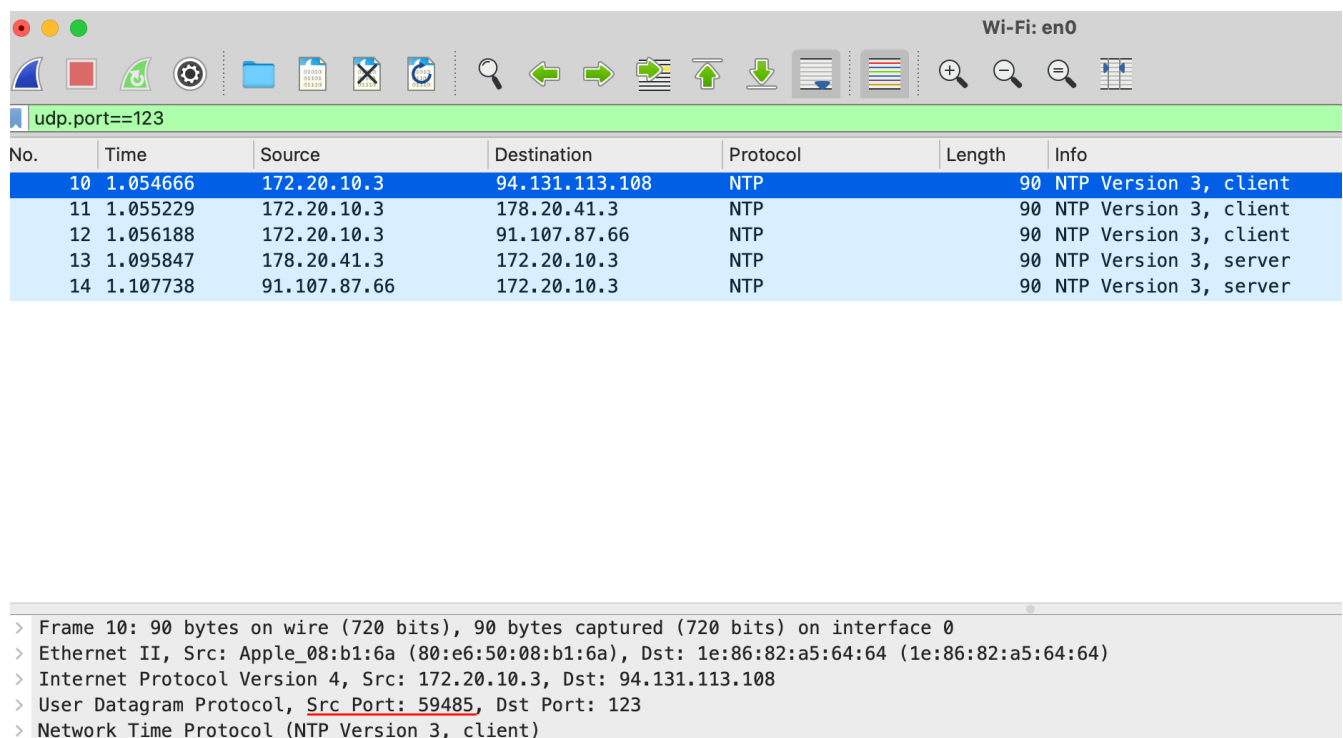
except Exception as e:
    print ("Socket error: " + str(e))
```

Запустим код. У меня далеко не всегда удавалось получить 3 ответа, а вот 2 два ответа получаю регулярно.

```
(venv) ScrumBook:src ilya2$ python3 udp-client-1-3host.py
Response received from: ('77.244.214.141', 123)
    Time=Fri Oct 18 16:05:51 2024
Response received from: ('91.206.16.3', 123)
    Time=Fri Oct 18 16:05:51 2024
Socket error: socket recv broken
(venv) ScrumBook:src ilya2$
```

Figure 2. Результат отправки SNTP-запроса на 3 различных SNTP-сервера.

Если посмотреть пакеты в сниффере (например, в Wireshark), то мы обнаружим, что все запросы отправились с одного UDP порта. У меня это 59 485, как показано на рисунке ниже.



No.	Time	Source	Destination	Protocol	Length	Info
10	1.054666	172.20.10.3	94.131.113.108	NTP	90	NTP Version 3, client
11	1.055229	172.20.10.3	178.20.41.3	NTP	90	NTP Version 3, client
12	1.056188	172.20.10.3	91.107.87.66	NTP	90	NTP Version 3, client
13	1.095847	178.20.41.3	172.20.10.3	NTP	90	NTP Version 3, server
14	1.107738	91.107.87.66	172.20.10.3	NTP	90	NTP Version 3, server

> Frame 10: 90 bytes on wire (720 bits), 90 bytes captured (720 bits) on interface 0	
> Ethernet II, Src: Apple_08:b1:6a (80:e6:50:08:b1:6a), Dst: 1e:86:82:a5:64:64 (1e:86:82:a5:64:64)	
> Internet Protocol Version 4, Src: 172.20.10.3, Dst: 94.131.113.108	
> User Datagram Protocol, Src Port: 59485, Dst Port: 123	
> Network Time Protocol (NTP Version 3, client)	

Figure 3. Результат работы Wireshark.

В коде мы сразу отправляем 3 SNTP-запроса разным хостам с использованием функции `sendto`. А раз мы отправили 3 запроса, то ожидаем 3 ответа. Не факт, что все они придут, но мы их ждем! Это означает, что на наш UDP порт (в данном случае 59 485) может прийти три SNTP-ответа (от 1.ru.pool.ntp.org, 2.ru.pool.ntp.org и от 3.ru.pool.ntp.org).

```
for i in range(3):
    rdy = select.select([client], [], [], 0.9)

    if not rdy[0]:
        raise RuntimeError("socket recv broken")

    data, address = client.recvfrom(1024)

    if data:
```

```
print ('Response received from:', address)
t = struct.unpack( '!12I', data)[8]
t -= TIME1970
print ('\tTime=%s' % time.ctime(t))
```

Поэтому в цикле вызываем функцию `select` и ждем ответа. Если ответ пришел, выводим информацию об отправителе и его точное время. А если ответ не пришел, вызываем исключение.

UDP-сокеты позволяют отправлять пакеты множеству хостов и затем получать от них ответы.

UDP-клиент и метод `connect`

Как известно, протокол UDP не устанавливает соединения и не заботится о надежной доставке данных. Поэтому, при работе с UDP-сокетом, обычно, вызов метода `connect` не производится.

При вызове метода `sendto` указывается IP-адрес и порт, на который нужно отправить данные. И именно в этот момент наша ОС выбирает порт источника, который будет указан в UDP-пакете. И даже в этот момент никакого соединения не устанавливается.

Чтобы в этом убедиться, проведем эксперимент - перед вызовом `sendto` вызовем функцию `sleep` из модуля `time`. Вызов функции `sleep` заставит программу уснуть на указанное число секунд, в моем случае, это 20. А в это время мы посмотрим состояние наших сокетов используя утилиту `netstat`

```
import socket
import struct
import time
import select

NTP_SERVER = "2.ru.pool.ntp.org"
PORT = 123
TIME1970 = 2208988800

client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

data = '\x1b' + 47 * '\0'

try:
    time.sleep(20)
    client.sendto( data.encode('utf-8'), (NTP_SERVER, PORT))
    rdy = select.select([client], [], [], 0.9)

    if not rdy[0]:
        raise RuntimeError("socket recv broken")

    data, address = client.recvfrom(1024)
```

```

if data:
    print ('Response received from:', address)
    t = struct.unpack( '!12I', data)[8]
    t -= TIME1970
    print ('\tTime=%s' % time.ctime(t))
except Exception as e:
    print ("Socket error: " + str(e))

```

Запустим программу, и пока она спит, посмотрим список открытых соединений на нашем хосте.

NOTE

Если у вас ОС Linux или MacOS, то список открытых соединений можно посмотреть командой `netstat -an`. Если у вас Windows, то наберите в консоли `netstat -n`

У меня на хосте (MacOS) очень много открытых соединений и чтобы не искать нужное, я отфильтрую их по строке 123 командой `grep`. Фильтрация происходит по 123, так как именно на этот UDP порт мы отправляем SNTP-запросы.



```

ScrumBook:~ ilya2$ netstat -an | grep 123
tcp6      0      0  *.61233      *.*          LISTEN
tcp4      0      0  *.61233      *.*          LISTEN
ScrumBook:~ ilya2$

```

Figure 4. Вывод команды `netstat` в MacOS


Как видно, никаких открытых соединений. А теперь давайте вызовем функцию `sleep` после функции `sendto`.

```

try:
    client.sendto( data.encode('utf-8'), (NTP_SERVER, PORT))
    time.sleep(2)
    rdy = select.select([client], [], [], 0.9)

```

На этот раз поспим всего 2 секунды. Снова выполним программу и пока она спит, быстро посмотрим список открытых соединений.



```

ScrumBook:~ ilya2$ netstat -an | grep 123
tcp6      0      0  *.61233      *.*          LISTEN
tcp4      0      0  *.61233      *.*          LISTEN
ScrumBook:~ ilya2$

```

Figure 5. Вывод команды `netstat` в MacOS

Вывод команды `netstat` оказался аналогичным первому, никаких открытых соединений нет. Это логично, ведь UDP не открывает соединения, а когда мы отправляем данные с помощью функции `sendto`, то ОС просто запоминает IP-адрес и порт назначения и выбранный случайным образом порт источника.

А вот если вызывать метод `connect` на UDP сокет, то ОС:

- сразу выделит порт источника для будущих UDP-пакетов
- добавит к сокету состояние соединения (установлено)

Соединение будет считаться установленным, так как для его установки не требуется отправка каких-либо пакетов.

Зачем это нужно?

Хоть это и не главное достоинство, но после вызова функции `connect` на UDP можно вызывать функции `send` и `recv` вместо `sendto` и `recvfrom`.

```
import socket
import struct
import time
import select

NTP_SERVER = "3.ru.pool.ntp.org"
PORT = 123
TIME1970 = 2208988800

client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

data = '\x1b' + 47 * '\0'

try:
    client.connect((NTP_SERVER, PORT))
    client.send(data.encode('utf-8'))

    rdy = select.select([client], [], [client], 0.9)

    if not rdy[0]:
        raise RuntimeError("socket recv broken")

    data = client.recv(1024)

    if data:
        t = struct.unpack('!12I', data)[8]
        t -= TIME1970
        print('\tTime=%s' % time.ctime(t))
except Exception as e:
    print("Socket error: " + str(e))
```

Запустив код мы увидим, что он работает нормально. Если SNTP-ответ доходит до нас, то мы просто печатает точно время, как обычно. А если нет, то получаем сообщение об ошибке.

Теперь с UDP сокетом можно работать также, как и с TCP. Если вы не собираетесь в рамках одного сокета работать с несколькими хостами, то смело вызывайте функцию `connect` вы используйте привычные функции `send` и `recv`.

Другим большим преимуществом вызову функции `connect` на UDP-сокеты является обработка сетевых ошибок.

Когда TCP-клиент пытается установить соединение на закрытый порт, то в ответ он получает TCP-пакет с флагом RST. Таким образом, при вызове функции `connect` у TCP-клиента можно сразу сделать вывод о невозможности связаться с удаленной стороной и не отправлять данные. Или, удаленная сторона может закрыть соединение и тогда функция `recv` на TCP-клиенте вернет 0 байт.

А что делать в случае с UDP? Например, мы написали UDP-клиента для получения точного времени с SNTP серверов. Отправили SNTP-запрос, а в ответ тишина. Как реагировать на отсутствие ответов:

- SNTP-запрос не дошел до сервера?
- SNTP-запрос не дошел от сервера до нас?
- нет сервера, который слушает порт 123 (UDP)?

В первых двух случаях мы можем повторить отправку данных. А вот в последнем случае нет смысла повторно отправлять данные. Даже если они дойдут до удаленного хоста, никто не будет их обрабатывать.

Да, UDP не устанавливает соединение. И для того, чтобы хоть как-то определить отсутствие готовности удаленной стороны принимать данные используется ICMP сообщения. В частности, если хост получает UDP пакет на порт, который никто не слушает, то ОС в ответ отправляет ICMP сообщение тип=3, код=3 - Destination Port Unreachable.

Получив такое ICMP сообщение, ОС изменит состояние соединения и при попытке отправить или прочитать данные из сокета мы получим ошибку.

NOTE

Еще раз обратите внимание, что соединение - это просто некоторое состояние сокета.

Давайте воспроизведем ошибку. Отправим SNTP-запрос за закрытый порт. Например, на хост `miminet.ru` и порт 125.

```
import socket
import struct
import time
import select

NTP_SERVER = "miminet.ru"
PORT = 125
```

```

TIME1970 = 2208988800

client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

data = '\x1b' + 47 * '\0'

try:
    client.connect((NTP_SERVER, PORT))
    client.send( data.encode('utf-8'))

    rdy = select.select([client], [], [], 0.9)

    if not rdy[0]:
        raise RuntimeError("socket recv broken")

    data = client.recv(1024)

    if data:
        t = struct.unpack( '!12I', data)[8]
        t -= TIME1970
        print ('\tTime=%s' % time.ctime(t))
except Exception as e:
    print ("Socket error: " + str(e))

```

Теперь, если мы запустим код то мы получим сообщение **Connection refused**, как показано ниже.

NOTE

Вы можете не получить это сообщение с первого раза, если по пути потеряется UDP или ICMP пакет. Попробуйте запустить программу несколько раз.

```

(venv) ScrumBook:src ilya2$ python3 udp-client-3-connect-icmp.py
Socket error: [Errno 61] Connection refused
(venv) ScrumBook:src ilya2$

```

Если во время выполнения программу запустить сниффер Wireshark, то мы увидим ICMP пакет "Destination Port unreachable". И внутри этого сообщения будет находиться наш UDP пакет.

NOTE

Внутри ICMP сообщений с ошибками всегда находится пакет, который вызвал эту ошибку. Таким образом, получив такое ICMP сообщение ОС всегда сможет понять, какой именно пакет вызвал эту ошибку и отреагировать соответствующим образом. В нашем случае ОС установит состояние соединения у нашего сокета на "закрытый".

[illegible]

А вот еще один пример, UDP-клиенту запрещено отправлять UDP пакеты на внешние IP-адреса. При попытке отправить любой UDP пакет в ответ приходит ICMP сообщение тип=3, код=13 - Communication Administratively Prohibited. Оно также влияет на состояние сокета и мы получаем ошибку **Connection refused**.

```
> Frame 44: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface 0
> Ethernet II, Src: Routerbo_a2:54:30 (d4:ca:6d:a2:54:30), Dst: Apple_08:b1:6a (80:e6:50:08:b1:6a)
> Internet Protocol Version 4, Src: 195.19.224.65, Dst: 172.22.241.4
> Internet Control Message Protocol
  Type: 3 (Destination unreachable)
  Code: 13 (Communication administratively filtered)
  Checksum: 0xebb1 [correct]
  [Checksum Status: Good]
  Unused: 00000000
> Internet Protocol Version 4, Src: 172.22.241.4, Dst: 51.250.2.96
> User Datagram Protocol. Src Port: 58553. Dst Port: 125
```

Если вы используете UDP сокет для работы с одним удаленным приложением, то я еще раз рекомендую использовать функцию `connect`. Это позволит вам обнаруживать и реагировать на ошибки в сети.

Сокеты

Для работы с сетью на низком уровне используются сокеты (socket). Сокет - это название программного интерфейса (API) для обеспечения обмена данными между процессами. Процессы при этом могут исполняться как на одном хосте, так и на различных.

Хорошей аналогией сокета является файловый дескриптор - некоторая абстракция, для работы с файлами в ОС. Вы можете открыть или закрыть файловый дескриптор, можете прочитать или записать туда данные и т.д.

Использование сокета у клиента немного отличается от сервера. Во время разработки сервера вы увидите эту разницу.

NOTE

Подробности по использованию сокетов в Python можно найти [в официальной документации](#).

ТСР-клиент

Давайте напишем первого сетевого клиента, который будет подключаться к серверу miminet.ru на порт 80 (ТСР), отправлять HTTP-запрос, получать HTTP-ответ и печатать содержимое ответа в консоль. Схема работы показана на рисунке ниже.

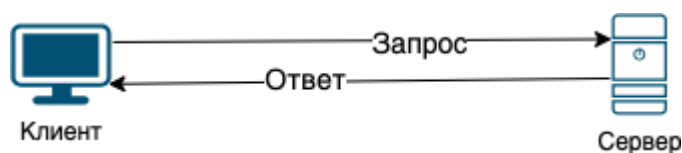


Figure 8. Схема работы ТСР-клиента.

NOTE

HTTP (HyperText Transfer Protocol) - протокол для коммуникации с веб-сервером. HTTP-запрос - запрос отправляемый клиентом на веб-сервер. В ответ на такой запрос веб-сервер отправляет данные (HTML страницу, JS код, картинки, видео и т.д.) или отправляет информационное сообщение.

NOTE

По умолчанию, веб-сервер для работы использует порт 80 (ТСР).

Нам необходимо выполнить следующие действия:

1. Создать ТСР сокет.
2. Подключиться к веб-серверу miminet.ru на порт 80.
3. Отправить HTTP-запрос.
4. Получить ответ и вывести его на экран.

```
#!/usr/bin/python

import socket
```

```

HOST = "miminet.ru"
PORT = 80                                     # HTTP port

ip_addr = socket.gethostbyname(HOST)          # Get IP address by name
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Create TCP socket
s.connect((ip_addr, PORT))                    # Connect to miminet.ru
s.send(b'GET / HTTP/1.0\n\n')                 # Send HTTP-request
data = s.recv(4096)                           # Recieve data from miminet.ru
print(data)                                   # Print data to console

```

Вы можете скопировать этот код и попробовать выполнить его у себя. В результате его работы в консоль должно вывестись следующее сообщение:

```

(venv) ScrumBook:src ilya2$ python tcp-client-1.py
b'HTTP/1.1 301 Moved Permanently\r\nServer: nginx/1.25.4\r\nDate: Wed, 16 Oct 2024
08:40:37 GMT\r\nContent-Type: text/html\r\nContent-Length: 169\r\nConnection:
close\r\nLocation: https://miminet.ru/\r\n\r\n<html>\r\n<head><title>301 Moved
Permanently</title></head>\r\n<body>\r\n<center><h1>301 Moved
Permanently</h1></center>\r\n<hr><center>nginx/1.25.4</center>\r\n</body>\r\n</html>\r
\n'

```

Я постарался прокомментировать каждую строку, но давайте подробнее разберем, что мы делаем и что происходит

```
import socket
```

Импортируем модуль `socket`. Как раз он нам позволит работать с сокетами.

```
ip_addr = socket.gethostbyname(HOST)
```

Для создания TCP соединения необходимо знать IP-адрес хоста и порт, на который мы хотим подключиться. Пока мы знаем только имя хоста `miminet.ru` и порт `80`. Функция `gethostbyname` позволяет по имени определить IP-адрес хоста. Т.е. по имени `miminet.ru` она вернет IP-адрес хоста. Сохраним его в переменную `ip_addr`.

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Данный код создает TCP сокет и сохраняет его в переменную `s`. Функция `socket` позволяет создавать не только TCP или UDP сокеты. Она принимает два обязательных параметра:

- `socket.AF_INET` - когда мы хотим открыть сокет из семейства протоколов TCP/IP
- `socket.SOCK_STREAM` - для создания TCP-сокета.

И так, сокет создан, IP-адрес и порт известны, можем подключаться.

```
s.connect((ip_addr, PORT))
```

Функция `connect` устанавливает TCP соединения (3-х разовое рукопожатие SYN, SYN+ACK, ACK) на заданный IP-адрес и порт. Обратите внимание, что функция `connect` принимает только один порт - это порт назначения. Т.е. порт, на который мы устанавливаем соединение. У клиентов (тот кто иницирует установку соединения) порт источника выбирается случайным образом.

```
s.send(b'GET / HTTP/1.0\n\n')
```

Функция `send` отправляет данные в установленное TCP соединение. В данном случае мы отправили HTTP-запрос на веб-сервер `miminet.ru`. HTTP - это текстовый протокол, поэтому, запрос можно написать прямым текстом. Два подряд идущие перевода строки `\n\n` означают окончание HTTP-запроса. В этом HTTP-запросе мы хотим получить главную страницу `miminet.ru`.

```
data = s.recv(4096)
```

Получаем данные из сокета. Функция `recv` по умолчанию работает в блокирующем режиме. Это означает, что вызвав эту функцию программа будет ожидать, пока не придут данные. Так как заранее не известно, сколько данных придет, указал 4096 байт. Если данных придет меньше, вернут их, если придет больше, вернут не более 4096 байт. Полученные данные сохраняем в переменную `data`.

```
print(data)
```

Печатаем полученные данные в консоль.

NOTE

На самом деле можно опустить вызов функции `gethostbyname`, так как функция `connect` может самостоятельно по имени определить IP адрес. Я решил оставить `gethostbyname` для наглядности.

Обработка ошибок

Код из примера выше годится только для примера. Не трудно заметить, что там нет обработки ошибок. Давайте добавим обработку ошибок и посмотрим, что может пойти не так.

```
#!/usr/bin/python

import socket

HOST = "miminet.ru"
```

```

PORT = 80

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    s.connect((HOST, PORT))
    s.send(b'GET / HTTP/1.0\n\n')
    data = s.recv(4096)
    print(data)
except Exception as e:
    print ("Socket error: " + str(e))

```

Ошибка преобразования имени в IP адрес

Попробуем вместо имени хоста `miminet.ru` подставить несуществующее имя, например, `miminet.rus`.

```
HOST = "miminet.rus"
```

Для этого изменим имя хоста в переменной `HOST` и запустим наш код. В результате мы получим ошибку:

```

(venv) ScrumBook:src ilya2$ python tcp-client-2-hostname.py
Socket error: [Errno 8] nodename nor servname provided, or not known

```

Она означает, что нам не удалось по имени хоста определить IP-адрес. Дальнейшая установка соединения бессмысленно, так как мы не знаем IP-адрес хоста.

NOTE Для воспроизведения следующих ошибок верните имя в переменной `HOST` обратно на `miminet.ru`

Ошибка подключения

Для воспроизведения ошибки подключения поменяем переменную `PORT` с 80 на 81 (можно и 81 и многие другие порты, которые закрыты). Запустим наш код и увидим ошибку подключения:

```

(venv) ScrumBook:src ilya2$ python tcp-client-2-port81.py
Socket error: [Errno 61] Connection refused

```

Данная ошибка сообщает, что на удаленной стороне нет программы, которая готова работать на указанном порту. В моем случае - это порт 81. Нет смысла продолжать выполнять программу и пытаться отправить данные. Ошибка появляется, когда во время установки TCP соединения клиент получает пакет с флагом RST. Из хорошего, данная ошибка появляется быстро. Т.е. наш хост отправил SYN пакет, в ответ получил RST и

сообщил об этом нам.

А теперь попробуем установить TCP соединение на порт, который не будет отвечать пакетом с флагом RST. Посмотрим, как себя поведет наша программа. Для этого на сервере `miminet.ru` настроен фаервол, который отбрасывает все входящие TCP пакеты с портом назначения равным 8000.

Поменяем переменную `PORT` на 8000 и запустим нашу программу. После длительного ожидания появляется ошибка:

```
(venv) ScrumBook:src ilya2$ python tcp-client-2-port8000-1.py
Socket error: [Errno 60] Operation timed out
```

Ошибка означает, что не удалось установить TCP соединение. Такую ошибку можно наблюдать, когда пакеты по какой-то причине не доходят до сервера. Либо они блокируются фаерволом, либо сервер просто выключен.

Это неприятная ошибка! Сокет и все его функции, включая `connect`, по умолчанию работают в блокирующем режиме. Это когда программа вызывает функцию и ждет, пока эта функция не завершит свою работу. И когда пакеты во время установки соединения вот так теряются, то вся программа зависает. В моем случае программа зависла на 75 секунд.

```
(venv) ScrumBook:src ilya2$ time python tcp-client-2-port8000-1.py
Socket error: [Errno 60] Operation timed out

real    1m15.779s
user    0m0.028s
sys     0m0.012s
```

NOTE

Если у вас Linux или MacOS, то для замера времени исполнения программы, перед запуском напишите `time`. Общее время исполнения программы будет отображаться в строке `real`.

Чтобы решить проблему с подвисанием, можно запустить работу с сокетом в отдельном потоке. Но, даже в отдельном потоке установка соединения может происходить аж 70 секунд. Это очень долго! Современные сети работают на много быстрее, чтобы ждать столько времени перед тем, как будет принято решение о невозможности установить соединение. Для уменьшения таймаута воспользуемся функцией `settimeout`.

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(5)
```

Сразу после создания сокета установим таймаут на блокирующие операции в 5 секунд. Теперь функция `connect` ожидает всего 5 секунд, после чего сообщает об ошибке и завершает свою работу.

```
(venv) ScrumBook:src ilya2$ time python tcp-client-2-port8000-2.py
```

```
Socket error: timed out
```

```
real    0m5.055s
```

```
user    0m0.032s
```

```
sys     0m0.013s
```

NOTE

Установка таймаута в 0 переведет сокет в неблокирующий режим. В этом случае нужно будет поменять схему работы с сокетом. Об этом будет подробнее рассказано дальше по курсу.

При работе с сокетом я всегда рекомендую уменьшать таймауты до приемлемого значения. Современные компьютерные сети позволяют на много быстрее определить невозможность установки соединения и сообщить об этом.

Обработка отправки данных (send)

Функция `send` обычно отработывает без сбоев. Но вот что стоит о ней знать! Когда вы вызываете `send`, то данные не передаются приложению на другом конце сокета. Функция `send` только помещает данные в буфер для отправки. И все.

После того как функция `send` поместила данные на отправку, соединение может быть уже разорвано и, соответственно, никакие данные никуда не будут переданы. Учтите этот момент!

Обработка получения данных (recv)

Функция `recv` принимает один обязательный аргумент - это максимальное количество байт, которое можно вернуть. В случае ошибки функция `recv` вернёт 0 байт данных. Это будет означать, что соединение было закрыто и от туда больше ничего не может быть получено.

Особо внимание стоит обратить на работу функции `recv`. Функция `recv` - блокирующая функция и она будет ожидать данные вечно (либо пока соединение не будет закрыто). Теоретически, вызвав `recv` вы можете вечно ожидать, пока она что-то вернет.

Чтобы воспроизвести проблему с `recv` изменим наш код на следующий:

```
#!/usr/bin/python
```

```
import socket
```

```
HOST = "miminet.ru"
```

```
PORT = 80
```

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
s.settimeout(5)
```

```
try:
```

```
s.connect((HOST, PORT))
data = s.recv(4096)
print(data)
except Exception as e:
    print ("Socket error: " + str(e))
```

В этом коде мы убрали отправку HTTP-запроса (`s.send`) и сразу ожидаем данные. Запустите этот код. Программа будет ожидать данные до тех пор, пока сокет не будет закрыт или пока не истечет таймаут.

```
(venv) ScrumBook:src ilya2$ time python tcp-client-2-recv.py
Socket error: timed out

real    0m5.079s
user    0m0.028s
sys     0m0.011s
```

В этом случае ожидание длилось 5 секунд - время установленного таймаута.

ТСР-клиент с обработкой ошибок

Давайте посмотрим, как будет выглядеть наш ТСР-клиент с обработкой указанных ошибок:

```
#!/usr/bin/python

import socket

HOST = "miminet.ru"
PORT = 80

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(5)

try:
    s.connect((HOST, PORT))
    s.send(b'GET / HTTP/1.0\n\n')
    data = s.recv(4096)

    if not data:
        raise RuntimeError("socket connection broken")

    print(data)
except Exception as e:
    print ("Socket error: "+str(e))
```

```
s.settimeout(5)
```

Чтобы не ждать слишком долго блокирующих операций (при невозможности установить соединение и когда нечего читать в буфере приема).

```
if not data:
    raise RuntimeError("socket connection broken")
```

Проверяем результат работы функции `recv` и в случае ошибки сообщаем, что соединение было закрыто.

В целом, уже не плохо!

Правда, проблема с `recv` до конца не решена. Если данные не поступят, то `recv` будет их ждать 5 секунд.

Проверка доступности данных (select)

Можно работу с сокетами выделить в отдельный поток (`thread`) и не переживать о блокировке кода. Но, это перенос проблемы из одного места в другое. Для решения данной проблемы используется функция `select`.

Функция `select` позволяет проверить наличия данных в буфере, что дает возможность вызывать `recv` только тогда, когда буфер не пуст и избегать ненужных ожиданий. Ниже представлен код с использованием `select`

```
#!/usr/bin/python

import socket
import select

HOST = "miminet.ru"
PORT = 80

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(5)

try:
    s.connect((HOST, PORT))
    s.send(b'GET / HTTP/1.0\n\n')

    rdy = select.select([s], [], [], 2)
    if not rdy[0]:
        raise RuntimeError("no response")

    data = s.recv(4096)

    if not data:
        raise RuntimeError("socket connection broken")
```

```
print(data)

except Exception as e:
    print ("Socket error: "+str(e))
```

Функция `select` принимает 4 аргумент:

- список дескрипторов, готовых для чтения
- список дескрипторов, готовых для записи
- список дескрипторов которые в исключительном состоянии (exceptional condition)
- время ожидания (float)

```
rdy = select.select([s], [], [], 2)
if not rdy[0]:
    raise RuntimeError("no response")
```

Как и многие другие функции, `select` - блокирующая функция. Код выше означает - жать 2 секунды или пока в сокете `s` не появятся данные для чтения.

Проверка нужна для того, чтобы определить, функция `select` завершилась по таймауту (2 секунды) или появились данные для чтения.

NOTE

функция `select` работает с дескрипторами и ей все равно, это сокет, файловый дескриптор или дескриптор для ввода/вывода с консоли.

Таким образом, у нас получился следующий TCP-клиент:

- если все хорошо, то все хорошо
- если невозможно установить TCP соединение, сразу сообщаем об этом
- если TCP соединение не устанавливается 5 секунд (вместо 70), прекращаем работу с ошибкой
- если данные не приходят в ответ на запрос, ждем 2 секунды (вместо 5) и прекращаем работу.

Это уже на много лучше того, что было изначально.

NOTE

Вся сила функции `select` совсем не в том, что мы ждем всего 2 секунды. Она раскрывается при работе с множеством сокетов. Что бы для каждого сокета не выделять отдельный поток, используя `select` всю работу можно организовать в одном потоке.

Сервер

В архитектуре клиент-сервер под сервером понимается программа, готовая обрабатывать

данные на определенном IP-адресе и порту. Поэтому, перед началом работы, сервер регистрирует в ОС IP-адрес и порт для своей работы. Например:

- веб-сервер по умолчанию использует для работы порт 80 (TCP) или 443 (TCP) для безопасного соединения.
- почтовый сервер использует порт 25 (TCP)
- SSH - 22 (TCP)
- DNS - 53 (UDP)
- и т.д.

ТСР-сервер

Отличительной особенностью реализации сервера является открытие так называемого серверного сокета. Общая идея такая:

- создаем сокет (socket)
- регистрируем в ОС IP-адрес и порт для работы (bind)
- переводим сокет в режим ожидания соединения от клиентов (listening)
- при подключении нового клиента ОС создает еще один сокет для работы с подключившимся клиентом (ассерт). Этот сокет еще называют "клиентский сокет"
- выполняем обычную работу с клиентским сокетом (recv, send, select, close/shutdown).

Давайте напишем простой ТСР эхо-сервер и подробно разберём его устройство. ТСР эхо-сервер будет принимать подключение от клиента и отправлять в ответ все, что клиент пришлёт ему.

```
import socket

HOST = 'localhost'
PORT = 30000
data_payload = 2048
backlog = 5

# Create a TCP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Enable reuse address/port
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

print("Starting up echo server on %s port %s" % (HOST, PORT))
sock.bind((HOST, PORT))

# Listen to clients, backlog argument specifies the max no. of queued connections
sock.listen(backlog)

while True:
```

```

print("Waiting to receive message from client")
client, address = sock.accept()
print("Client connected from %s" % (address,))
data = client.recv(data_payload)
if data:
    print("Data: %s" % data)
    s = client.send(data)
    print("sent %s bytes back" % (s,))

    # end connection
client.close()

```

Разберем этот пример подробнее.

```

import socket

HOST = 'localhost'
PORT = 30000

```

Переменные HOST и PORT указывают, какой IP и порт будет слушать TCP-сервер. Как и клиент, сервер может работать только с конкретным IP адресом, поэтому, значение **localhost** в последствии превратиться в IP-адрес 127.0.0.1.

Когда сервер слушает только IP-адрес 127.0.0.1, то подключиться к нему можно только с этого же хоста. Многие современные сервера, например, PostgreSQL, можно настраивать (администрировать) подключившись через сокет. Чтобы кто-то случайно или намеренно не смог подключиться из внешней сети и начать настраивать ваш PostgreSQL сервер, по умолчанию, сервер открывает сокет для настройки на адресе 127.0.0.1. Таким образом он гарантирует, что подключиться к нему сможет только локальный пользователь.

NOTE

Другим примером является ваш домашний Wi-Fi маршрутизатор. Многие домашние Wi-Fi маршрутизаторы можно настраивать удаленно, через веб-интерфейс. В целях безопасности, подключиться к веб-интерфейсу, по умолчанию, можно только из внутренней сети. Т.е. веб-сервер слушает не 127.0.0.1, а интерфейс с IP-адресом внутренней сети. Таким образом, настраивать Wi-Fi маршрутизатор могут все, кто смог подключиться к нему через Wi-Fi.

```

data_payload = 2048
backlog = 5

```

Переменная **data_payload** содержит размер данных, которые мы будем читать от клиента. Если придет меньше, то хорошо, а если больше, то сервер прочитает первые 2048 байта и все. Этот TCP-эхо-сервер для примера и нет смысла читать данные большими объемами. 2048 байт будет вполне достаточно.

Переменная **backlog** указывает на размер очереди на подключение.

```
# Create a TCP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Создаем TCP-сокеты и сохраняем его в переменную sock.

```
# Enable reuse address/port
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

Используем функцию setsockopt для возможности переиспользовать IP-адрес и порт. Что это для чего будет немного дальше. А пока просто подметим это.

```
print("Starting up echo server on %s port %s" % (HOST, PORT))
sock.bind((HOST, PORT))
```

Функция **bind** сообщает ОС какой IP-адрес и порт будущий сервер хочет использовать для своей работы. В качестве IP-адреса можно указать и имя, например, **localhost**. Получив вместо IP-адреса имя, функция bind постарается сопоставить этому имени конкретный IP-адрес.

```
sock.listen(backlog)
```

Функция **listen** переведет сокет в режим ожидания соединений от клиентов. Именно этот вызов разрешает клиентам подключаться к нашему серверу. Параметр **backlog** сообщает о размере очереди на ожидания клиентов, которые могут успешно подключиться к нашему серверу и ожидать обработку.

```
while True:
    print("Waiting to receive message from client")
    client, address = sock.accept()
    print("Client connected from %s" % (address,))
    data = client.recv(data_payload)
    if data:
        print("Data: %s" % data)
        s = client.send(data)
        print("sent %s bytes back" % (s,))

    # end connection
    client.close()
```

А дальше идет вечный цикл. Предполагается, что любой сервер работает вечно, поэтому рано или поздно должен встретиться вечный цикл. В этом цикле наш TCP эхо-сервер производит основную работу.


```
client, address = sock.accept()
```

Ожидаем подключения нового клиента. Функция `accept` блокирующая и сервер будет находиться в ней до тех пор, пока не подключиться клиент. Как только клиент подключится, функция `accept` вернет два значения:

- сокет для работы с клиентом (клиентский сокет)
- IP-адрес и порт клиента.

```
data = client.recv(data_payload)
```

Функция `recv` ожидает данные от клиента. Как было сказано в начале, `recv` прочитает от клиента не более `data_payload` (2048) байт данных. Напомню, что функция `recv` блокирующая и тут мы тоже будем находиться до тех пор, пока клиент не пришлёт данные или пока сокет не будет закрыт.

```
if data:
    print("Data: %s" % data)
    s = client.send(data)
    print("sent %s bytes back" % (s,))
```

Если от клиента пришли данные, то мы их печатаем в консоль и отправляем (`send`) эти данные ему обратно, как настоящий эхо-сервер. После отправки данных печатаем в консоль количество байт отправленных данных.

```
client.close()
```

Вызываем функцию `close`, которая закроет клиентский сокет. После этого снова возвращаемся в начало нашего вечного цикла и ожидаем подключения следующего клиента.

Запуск TCP-сервера

Запустим наш TCP-сервер.

```
(venv) ScrumBook:src ilya2$ python tcp-server-1.py
Starting up echo server on localhost port 30000
Waiting to receive message from client
```

Если сервер запустился удачно, то мы увидим сообщение об этом. Теперь, давайте подключимся к нашему серверу из другой консоли. Для этого я использую утилиту `netcat` (`nc`).

NOTE

Для подключения к TCP-серверу я использую команду `netcat` (nc). В Ubuntu её можно установить командой `apt install netcat`. В MacOS она устанавливается командой `brew install netcat`.

При подключении через netcat нужно передать два параметра:

- IP-адрес или имя хоста для подключения. В нашем случае это будет localhost или можно написать 127.0.0.1.
- Номер порта для подключения. В нашем случае это порт 30000.

Ниже результат подключения к нашему серверу через утилиту netcat.

```
ScrumBook:~ ilya2$ nc localhost 30000
Hello TCP-server
Hello TCP-server
ScrumBook:~ ilya2$
```

После подключения я отправил строку Hello TCP-server. Вторая строка Hello TCP-server - это уже ответ от сервера. После получения ответа от сервера соединение было закрыто. Итого, наш сервер отработал, как и задумывалось.

```
(venv) ScrumBook:src ilya2$ python tcp-server-1.py
Starting up echo server on localhost port 30000
Waiting to receive message from client
Client connected from ('127.0.0.1', 57195)
Data: b'Hello TCP-server\n'
sent 17 bytes back
Waiting to receive message from client
```

Если посмотрим в консоль, где запущен TCP-сервер, то обнаружим вывод нескольких дополнительных сообщений:

- Client connected from ('127.0.0.1', 57195) - IP-адрес и порт клиента
- Data: b'Hello TCP-server\n' - строку, которую прислал клиент
- sent 17 bytes back - размер данных отправленных клиенту обратно
- Waiting to receive message from client - ожидание следующего клиента

Мы можем повторить подключение через `nc`, результат должен быть одинаков.

Итого, для создания сервера необходимо:

1. Создать сокет (socket).
2. Зарегистрировать в ОС IP-адрес и порт для работы сервера (bind).
3. Перевести сокет в состояние прослушивание (listen).
4. В вечном цикле:

- a. Ожидание подключения клиента (accept).
- b. Коммуникация с клиентом (recv/send).
- c. Закрытие сокета (close).

Обработка ошибок

Address already in use

Давайте чуть подробнее разберем необходимость опции `SO_REUSEADDR`. А для этого, давайте посмотрим, что нам покажет утилита `netstat`. Запустим наш сервер, как обычно, и посмотрим на список текущих сетевых соединений.

```
ScrumBook:~$ ssh ilya2$ netstat -ant | grep 30000
tcp4      0      0 127.0.0.1.30000      *.*      LISTEN
ScrumBook:~$ ssh ilya2$
```

NOTE

У меня на MacOS очень много сетевых соединений, поэтому я использую команду `grep`, чтобы отфильтровать только нужные мне.

Утилита `netstat` сообщает, что порт 30000 открыт и находится в состоянии `LISTEN`, т.е. готов к подключению новых клиентов.

А теперь давайте подключимся к нашему серверу, отправим сообщение `Hello` и снова посмотрим на список сетевых соединений.

```
ScrumBook:~$ ssh ilya2$ netstat -ant | grep 30000
tcp4      0      0 127.0.0.1.30000      *.*      LISTEN
ScrumBook:~$ ssh ilya2$ nc localhost 30000
Hello
Hello
ScrumBook:~$ ssh ilya2$ netstat -ant | grep 30000
tcp4      0      0 127.0.0.1.30000      *.*      LISTEN
tcp4      0      0 127.0.0.1.30000      127.0.0.1.58156  TIME_WAIT
ScrumBook:~$ ssh ilya2$
```

Мы увидим, что появился еще одно соединение на порту 30000, где IP-адрес источника 127.0.0.1 и порт 58156. Это соединение находится в состоянии `TIME_WAIT`. Это как раз наше соединение, по которому мы передали `Hello`. Состояние `TIME_WAIT` - это состояние, когда соединение уже закрыто, но ОС поддерживает его еще некоторое время, чтобы корректно обработать пакеты, которые по разным причинам могли задержаться в сети. Обычно этот таймаут составляет несколько десятков секунд или даже пару минут, зависит от настроек ОС.

Выполним следующие действия:

1. Запустим наш сервер (или он уже запущен).

2. Подключимся утилитой `nc`
3. Отправим любую строку, например, Hello.
4. Получим ответ и вместе с этим сервер закроет соединение.
5. Остановим сервер (CTRL+C).
6. И снова посмотрим на текущие соединения (netstat)

```
ScrumBook:~$ ssh ilya2$ netstat -ant | grep 30000
tcp4      0      0 127.0.0.1.30000      *.*      LISTEN
ScrumBook:~$ ssh ilya2$ nc localhost 30000
Hello
Hello
ScrumBook:~$ ssh ilya2$ netstat -ant | grep 30000
tcp4      0      0 127.0.0.1.30000      *.*      LISTEN
tcp4      0      0 127.0.0.1.30000      127.0.0.1.58824  TIME_WAIT
ScrumBook:~$ ssh ilya2$ netstat -ant | grep 30000
tcp4      0      0 127.0.0.1.30000      127.0.0.1.58824  TIME_WAIT
ScrumBook:~$ ssh ilya2$
```

Мы увидим, что осталось только последнее соединение в состоянии TIME_WAIT.

Закомментируйте строку с опцией SO_REUSEADDR.

```
# Enable reuse address/port
# sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

И попробуйте повторить все еще раз:

1. Запустите сервер
2. Подключитесь утилитой `nc` и отправьте любую строку
3. После ответа сервера, остановите сервер (CTRL+C).
4. Попробуйте снова запустить сервер.

При попытке запустить сервер вы получите ошибку `Address already in use`

```
Traceback (most recent call last):
  File "tcp-server-1.py", line 15, in <module>
    sock.bind((HOST, PORT))
OSError: [Errno 48] Address already in use
```

Пока есть хоть одно соединение на нашем порту (30000), мы не можем его использовать. Это не редкое состояние. Когда вы разрабатываете сервер, то вы будете очень часто выключать и включать его снова, для проверки работоспособности. И, чтобы не ждать таймаут TIME_WAIT перед повторным запуском сервера, можно установить флаг SO_REUSEADDR в 1. Этот флаг разрешает ОС переиспользовать сокет в состоянии TIME_WAIT.

Can't assign requested address

Сервер может слушать только назначенные IP-адреса в вашей ОС. Например, сейчас у меня на хосте назначены два IP-адреса:

- 127.0.0.1 (loopback)
- 192.168.1.131 (wi-fi)

```
ScrumBook:~ ilya2$ ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
    options=1203<RXCSUM,TXCSUM,TXSTATUS,SW_TIMESTAMP>
    inet 127.0.0.1 netmask 0xff000000
    inet6 ::1 prefixlen 128
    inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
    nd6 options=201<PERFORMNUD,DAD>
gif0: flags=8010<POINTOPOINT,MULTICAST> mtu 1280
stf0: flags=0<> mtu 1280
XHC20: flags=0<> mtu 0
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    options=400<CHANNEL_IO>
    ether 80:e6:50:08:b1:6a
    inet6 fe80::4c3:f15c:2b9a:3e5c%en0 prefixlen 64 secured scopeid 0x5
    inet 192.168.1.131 netmask 0xffffffff00 broadcast 192.168.1.255
    nd6 options=201<PERFORMNUD,DAD>
    media: autoselect
    status: active
```

Figure 9. IP-адреса на моём MacOS.

Это означает, что я могу запустить сервер на адресе 127.0.0.1 или на 192.168.1.131 или на обоих сразу. При попытке зарегистрировать не ваш IP-адрес вы получите ошибку **Can't assign requested address**.

Для примера, давайте запустим наш TCP эхо-сервер на адресе 77.88.8.8. Для этого поменяем переменную HOST в нашем примере.

```
import socket

HOST = '77.88.8.8'
PORT = 30000
```

При попытке запустить такой сервер мы получим ошибку.

```
(venv) ScrumBook:src ilya2$ python tcp-server-3.py
Starting up echo server on 77.88.8.8 port 30000
Traceback (most recent call last):
  File "tcp-server-3.py", line 15, in <module>
    sock.bind((HOST, PORT))
OSError: [Errno 49] Can't assign requested address
```

Если вы хотите принимать соединения на любой из доступным вам сетевых интерфейсов, то передайте пустое поле HOST.

```
import socket
```

```
HOST = ''
```

```
PORT = 30000
```

Запустив такой сервер и посмотреть netstat вы увидите примерно следующее

```
ScrumBook:~$ netstat -ant | grep 30000
tcp4      0      0  *.30000          *.*              LISTEN
ScrumBook:~$
```

Теперь рядом с номером порта 30000 стоит не 127.0.0.1, как было раньше, а символ *. Этот символ означает, что сервер готов принимать соединения на любой сетевой интерфейс в вашей ОС.

UDP-сервер

Давайте напишем простой UDP эхо-сервер и на примере разберем его работу.

```
import socket
```

```
HOST = ''
```

```
PORT = 30000
```

```
data_payload = 2048
```

```
# Create a TCP socket
```

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
print("Starting up UDP echo server on %s port %s" % (HOST, PORT))
```

```
sock.bind((HOST, PORT))
```

```
while True:
```

```
    print("Waiting to receive message from client")
```

```
    data, address = sock.recvfrom(data_payload)
```

```
    if not data:
```

```
        continue
```

```
    print("Received %s bytes from %s" % (len(data), address))
```

```
    s = sock.sendto(data, address)
```

```
    print("Sent %s bytes back" % (s,))
```

Он немного меньше, чем аналогичный TCP сервер:

- нет надобности использовать опцию `SO_REUSEADDR`, так как UDP сокет не может находиться в состоянии `TIME_WAIT`.

- нет надобности вызывать функцию `listen` и `accept`, так как UDP не ожидает соединений от клиентов.
- и не нужно в конце закрывать сокет функцией `close`.

Для работы UDP сервера нужно всего 3 действия:

1. Создать UDP сокет (socket).
2. Зарегистрировать IP-адрес и порт для работы (bind).
3. Обрабатывать запросы клиентов (recvfrom/sendto).

Запустим UDP эхо-сервер и попробуем с ним поработать.

```
(venv) ScrumBook:src ilya2$ python udp-server-1.py
Starting up UDP echo server on port 30000
Waiting to receive message from client
```

После запуска сервера в консоль выводится сообщение о том, что он готов к работе на порту 30000. Для взаимодействия с сервером воспользуемся утилитой `netcat` (nc).

NOTE

По умолчанию утилита `nc` устанавливает TCP соединение. Для отправки UDP пакетов на заданный адрес и порт, нужно установить флаг `-4u`. Например, `nc -4u localhost 30000`

```
ScrumBook:~ ilya2$ nc -4u localhost 30000
Hello, UDP server!
Hello, UDP server!
```

Как видно, отправив сообщение "Hello, UDP server!" мы получаем его в ответ. После получения сообщения мы можем дальше продолжать взаимодействовать с сервером.

NOTE

Обратите внимание, что UDP сервер не разрывает соединение после отправки ответа, как это было в TCP сервере. Еще раз напомним - UDP протокол не поддерживает соединений.

Так как UDP не устанавливает соединений, а работает с датаграммами, то с UDP сервером могут взаимодействовать сразу несколько клиентов. Чтобы в этом убедиться, запустите несколько (2-3) консолей и в каждой запустите команду `nc -4u localhost 30000`. После этого вы можете поочередно печатать в консолях строки и получать ответы на них.

```
data, address = sock.recvfrom(data_payload)
```

Функция `recvfrom` возвращает не только данные, но и IP-адрес и порт клиента. Этого как раз достаточно, чтобы отправить клиенту ответ используя функцию `sendto`.

```
s = sock.sendto(data, address)
```

Многопоточный TCP-сервер

Однопоточный TCP-сервер хорош только для примера. В реальности, конечно, TCP-сервер всегда многопоточный и готов одновременно обслуживать несколько клиентов. Разберем несколько способов создания многопоточного TCP-сервера.

Многопоточный TCP-сервер (fork)

В основе многопоточного TCP-сервера на fork лежит вызов функции `fork()`. `Fork()` - это системный вызов в UNIX-подобных ОС (в ОС windows он эмулируется). Вызова `fork()` создает в ОС идентичный (полная копия процесса, включая состояния регистров процессора, открытых дескрипторов, памяти и фгалов) процесс (child/ребенок), который начинает свое выполнение со следующей команды сразу после `fork`.

NOTE | Более подробно про `fork()` - <https://man7.org/linux/man-pages/man2/fork.2.html>

Общая идея работы многопоточного сервера через `fork()` такая:

- Родительский процесс работает с серверным сокетом и ожидает подключения новых клиентов
- При подключении клиента родительский процесс создает дочерний процесс
- Дочерний процесс занимается обслуживанием клиента, а родительский, в это время, ожидает подключения нового клиента.

Ниже представлен пример многопоточного TCP эхо-сервера с использованием `fork()`.

```
#!/usr/bin/python

import socket
import os
import sys
import signal

HOST = 'localhost'
PORT = 30001
data_payload = 2048
backlog = 5

def sigchld_handler(*args):
    pid, exit_code = os.wait()
    print ("Child pid %d exiting with code %d" % (pid, exit_code//256))

def client_handler(client):
```



```

while True:
    data = client.recv(data_payload)

    if not data:
        break

    print("Data: %s" % data)
    s = client.send(data)
    print("sent %s bytes back" % (s,))

# End connection
client.close()

signal.signal(signal.SIGCHLD, sigchld_handler)

# Create a TCP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Enable reuse address/port
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

print("Starting up echo server on %s port %s" % (HOST, PORT))
sock.bind((HOST, PORT))

# Listen to clients, backlog argument specifies the max no. of queued connections
sock.listen(backlog)

while True:
    print("Waiting to receive message from client")
    client, address = sock.accept()
    print("Client connected from %s" % (address,))

    # Create new child
    p = os.fork()

    # We're child?
    if not p:
        client_handler(client)

    # Exit from child
    # Don't go to while True!
    sys.exit(1)

```

Запустим этот TCP эхо-сервер и проверим, на сколько он справляется с задачей. После запуска сервера он сообщит о готовности к работе, как показано ниже.

```

(venv) ScrumBook:src ilya2$ python tcp-server-1-fork.py
Starting up echo server on localhost port 30001

```

Waiting to receive message from client

Поочередно запустите утилиту `nc localhost 30001` из двух консолей. И попробуйте отправлять строки из первой и из второй консоли. У меня получается это сделать без проблем и задержек. Другими словами, TCP-сервер успешно обслуживает двух клиентов. Можете запустить еще несколько консолей, он успешно справится с десятками подобных запросов.

```
ScrumBook:~ ilya2$ nc localhost 30001
Hello TCP-server with fork
Hello TCP-server with fork
```

А теперь давайте разберемся, как все это работает.

NOTE

Считается, что вы уже понимаете основные моменты в работе TCP-сервера и разбор примера будет касаться только многопоточной реализации.

```
def sigchld_handler(*args):
    pid, exit_code = os.wait()
    print ("Child pid %d exiting with code %d" % (pid, exit_code//256))
```

Функция `sigchld_handler` - обработчик сигнала SIGCHLD. Когда наша программа вызовет функцию `fork()`, то в системе появится идентичный процесс. Этот новый процесс называют дочерний, а тот кто его породил - родительский. Во время завершения работы дочерний процесс сообщает родительскому свой код возврата (статус выхода). Этот код возврата будет находиться в памяти ОС до тех пор, пока родительский процесс его не прочитает. Чтобы родительский процесс понял, что один из его детей завершил свою работу, система посылает родительскому процессу сигнал CHILD (SIGCHLD).

Другими словами, при завершении дочернего процесса ОС отправляет родительскому процессу сигнал SIGCHLD и держит в памяти код возврата ребенка.

По умолчанию, у сигнала SIGCHLD стоит пустой обработчик и родительский процесс ничего не делает при его получении. Поэтому, мы поменяем обработчик сигнала SIGCHLD на свой.

```
signal.signal(signal.SIGCHLD, sigchld_handler)
```

Наша функция `sigchld_handler` вызывает функцию `os.wait`, которая вернет идентификатор завершившегося дочернего процесса (`pid`) и его код возврата (`exit_code`).

```
def client_handler(client):

    while True:
        data = client.recv(data_payload)
```

```

    if not data:
        break

    print("Data: %s" % data)
    s = client.send(data)
    print("sent %s bytes back" % (s,))

# End connection
client.close()

```

Функция `client_handler` будет заниматься обработкой клиента.

```
data = client.recv(data_payload)
```

В ней мы в вечном цикле ожидаем данных от клиента. Если данные пришли, отправляем их клиенту обратно и снова ждем данные от клиента. Очень простой эхо-сервер.

```
s = client.send(data)
```

Если клиент закроет соединение, то функция `recv` вернет пустое значение, мы выйдем из вечного цикла, закроем клиентское соединение со своей стороны и выйдем из функции.

```

# Create new child
p = os.fork()

# We're child?
if not p:
    client_handler(client)

# Exit from child
# Don't go to while True!
sys.exit(1)

```

А это основной код родительского процесса. Функция `fork` создает дочерний процесс и начинает его выполнение со следующей команды после вызова `fork`. Чтобы понять, где-кто, функция `fork` вернет родителю идентификатор (pid) ребенка, а ребенку 0.

Поэтому, если мы родитель, то пропускаем всю работу с клиентским сокетом и возвращаемся в функцию `accept`. Если мы ребенок, то вызываем функцию для работы с клиентом `client_handler`. По её завершению завершаем и весь наш дочерний процесс (`sys.exit(1)`).

Используя функцию `fork` можно очень просто реализовать многопоточный сервер.

Зомби-процессы

Если разработчик забудет или неверно реализует обработку сигнала SIGCHLD, то это приводит к появлению в системе так называемых зомби-процессов. Зомби-процесс - это завершившийся дочерний процесс, чей код возврата еще никто не прочитал. Зомби его называют потому, что он уже закончил работу и больше ничего делать не будет (по сути мертвый), но занимает pid и числится в списке процессов.

Давайте проведем небольшой эксперимент. Закомментируем строку с регистрацией нашего обработчика сигнала SIGCHLD.

```
#signal.signal(signal.SIGCHLD, sigchld_handler)
```

Запустим наш эхо-сервер, несколько раз подключимся к нему утилитой **netcat** и отправим 1-2 строки (не важно сколько строк, главное, подключиться и отключиться, чтобы был создан дочерний процесс).

```
ScrumBook:~ ilya2$ nc localhost 30001
Hello TCP-server with fork
Hello TCP-server with fork
^C
ScrumBook:~ ilya2$ nc localhost 30001
Hello
Hello
^C
ScrumBook:~ ilya2$ nc localhost 30001
Privet
Privet
^C
ScrumBook:~ ilya2$ nc localhost 30001
Hi, my name is...
Hi, my name is...
^C
ScrumBook:~ ilya2$ nc localhost 30001
```

А теперь посмотрим на список процессов.

NOTE

Если у вас ОС Linux, то это можно сделать с помощью команды **ps** с флагами **axf** (**ps axf**). Под MacOS лучше установить утилиту **pstree** через **brew** (**brew install pstree**).

```
| \--= 49798 ilya2
| /System/Library/Frameworks/Python.framework/Versions/2.7/Resources/Python.app/Contents
| /MacOS/Python tcp-server-1-fork-zombie.py
| |--- 49810 ilya2 (Python)
| |--- 49812 ilya2 (Python)
| |--- 49815 ilya2 (Python)
```

Как видно, у родительского процесса tcp-server-1-fork-zombie.py висит 4 дочерних процесса. Если посмотреть статусы этих процессов, то видно, что все дочерние процессы в состоянии Z+ (из документации по утилите ps "Marks a dead process (a 'zombie')")

```
49798 s000 S+      0:00.09
/System/Library/Frameworks/Python.framework/Versions/2.7/Resources/Python.app/Contents
/MacOS/Python tcp-server-1-fork-zombie.py
49810 s000 Z+      0:00.00 (Python)
49812 s000 Z+      0:00.00 (Python)
49815 s000 Z+      0:00.00 (Python)
52502 s000 Z+      0:00.00 (Python)
```

NOTE

Так как я запускаю все примеры на MacOS, то для просмотра статуса процесса мне приходится использовать утилиту `ps`. А для рисования древовидной структуры `pstree`. В ОС Linux утилита `ps` умеет рисовать древовидную структуру процессов и отображать их статусы сразу.

В процессе работы такого сервера в системе будут множиться зомби процессы. Лучше этого не допускать. А для этого не забывайте про сигнал SIGCHLD.

Межпроцессное взаимодействие

Функция `fork` создает отдельный процесс. Если во время работы вам потребуется обмениваться данными между обработчиками клиентов или между дочерним и родительским процессом, то вам придется использовать что-то для межпроцессного взаимодействия (пайп, сокет, файл, БД).

Многопоточный ТСП-сервер (threads/потоки/нити)

В основе многопоточного сервера на потоках (threads) лежит возможность запускать несколько потоков исполнения кода в рамках одного процесса. Ниже представлен наш ТСП эхо-сервер на потоках.

```
#!/usr/bin/python

import socket
import threading

HOST = 'localhost'
PORT = 30002
data_payload = 2048
backlog = 5

def client_handler(client):
```

```

while True:
    data = client.recv(data_payload)

    if not data:
        break

    print("Data: %s" % data)
    s = client.send(data)
    print("sent %s bytes back" % (s,))

    # End connection
    client.close()

# Create a TCP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Enable reuse address/port
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

print("Starting up echo server on %s port %s" % (HOST, PORT))
sock.bind((HOST, PORT))

# Listen to clients, backlog argument specifies the max no. of queued connections
sock.listen(backlog)

while True:
    print("Waiting to receive message from client")
    client, address = sock.accept()
    print("Client connected from %s" % (address,))

    # Create and start thread
    t = threading.Thread(target=client_handler, args=(client,))
    t.start()

```

Запустим этот TCP эхо-сервер и проверим его работоспособность. После запуска, как обычно, подключимся к нему из нескольких консолей и убедимся, что эхо-сервер способен обрабатывать несколько клиентов одновременно.

А когда вы все проверили, давайте разбираться, как все работает.

```

#!/usr/bin/python

import socket
import threading

```

Для работы с потоками в Python используется модуль `threading`.

```

def client_handler(client):

```

```

while True:
    data = client.recv(data_payload)

    if not data:
        break

    print("Data: %s" % data)
    s = client.send(data)
    print("sent %s bytes back" % (s,))

# End connection
client.close()

```

Наша функция `client_handler` для работы с клиентом. В начале она ожидает ввод от клиента. Если данные от клиента пришли, то она отправляет их ему обратно и снова начинает ждать данные от клиента. И так по кругу. А если функция `recv` вернулась без данных, значит соединение было закрыто и можно завершать работу. Вызываем функцию `close` и выходим из обработчика потока.

```

while True:
    print("Waiting to receive message from client")
    client, address = sock.accept()
    print("Client connected from %s" % (address,))

    # Create and start thread
    t = threading.Thread(target=client_handler, args=(client,))
    t.start()

```

Главный поток ожидает подключения нового клиента (`accept`). Когда клиент подключился, формирует новый поток и запускает его (`t.start()`).

В целом, это все!

NOTE При работе с потоками важно помнить про Global Interpreter Lock (GIL).

Такая реализация чуть попроще, чем через `fork()`, так как нет нужды обрабатывать `SIGCHLD`.

Реализации через `fork` и потоки имеют свои достоинства и недостатки. Например, реализация многопоточной работы на потоках позволяет использовать общую память и обмениваться данными между потоками. С другой стороны, из-за GIL все потоки реально будут исполняться поочередно.

Многопоточный TCP-сервер (потоки и `select`)

Если внимательно проанализировать реализации многопоточного TCP-сервера через потоки или вызов функции `fork`, то вы увидим, что большую часть времени обработчик клиента просто ждет данных от пользователя в функции `recv`. В таком случае, выделять

отдельный поток для работы с одним клиентом очень не эффективно.

Если мы не ожидаем большого количество клиентов, то для обработки сразу нескольких клиентов нам будет достаточно двух потоков:

- Один основной поток, который принимает подключения от новых клиентов
- Второй поток для работы со всеми подключенными клиентами.

Это можно добиться совмещая работу потоков и функции `select`.

Потоки могут общаться между собой используя общую память.

Raw socket (сырой/неструктурированный сокет)

Неструктурированные (raw) сокеты позволяют реализовать новые протоколы IPv4 в пространстве пользователя. Неструктурированный сокет предоставляет доступ к содержимому Ethernet фрейма. Можно самостоятельно формировать IP, TCP, UDP, ICMP и другие заголовки.

В отличие от обыкновенных сокетов, неструктурированные сокеты могут открыть только процессы с идентификатором эффективного пользователя, равным 0 (root), или имеющие мандат CAP_NET_RAW. Другими словами, открыть их может только root (Администратор) или пользователь, которому выдвли права на работу с такими сокетами.

Неструктурированные (raw) сокеты часто применяются для реализации снифера, IDS (Intrusion Detection System/Система Обнаружения Вторжения), для отправки ICMP пакетов (ping/pong) и так далее.

Raw-socket - ICMP

Используя raw сокет напишем свою простую реализацию утилиты `ping`:

1. Отправить ICMP эхо-запрос на определенный IP-адрес.
2. Получить ICMP эхо-ответ.
3. Провести расчет времени между запросом и ответом.
4. Напечатать результаты.

Программа получилась не очень маленькой, но не переживайте, сейчас мы разберемся в её работе.

```
#!/usr/bin/env python

import os
import socket
import struct
```



```

import select
import sys
import time

TARGET_HOST = "8.8.8.8"
ICMP_ECHO_REQUEST = 8
DEFAULT_TIMEOUT = 2
DEFAULT_COUNT = 4

class Pinger(object):

    def __init__(self, target_host, count=DEFAULT_COUNT, timeout=DEFAULT_TIMEOUT):
        self.target_host = target_host
        self.count = count
        self.timeout = timeout

    def do_checksum(self, source_string):
        """ Verify the packet integrity """
        sum = 0
        max_count = (len(source_string) / 2) * 2
        count = 0
        while count < max_count:
            # To make this program run with Python 2.7.x:
            # val = ord(source_string[count + 1])*256 + ord(source_string[count])
            # ### uncomment the above line, and comment out the below line.
            val = source_string[count + 1] * 256 + source_string[count]
            # In Python 3, indexing a bytes object returns an integer.
            # Hence, ord() is redundant.

            sum = sum + val
            sum = sum & 0xffffffff
            count = count + 2

        if max_count < len(source_string):
            sum = sum + ord(source_string[len(source_string) - 1])
            sum = sum & 0xffffffff

        sum = (sum >> 16) + (sum & 0xffff)
        sum = sum + (sum >> 16)
        answer = ~sum
        answer = answer & 0xffff
        answer = answer >> 8 | (answer << 8 & 0xff00)
        return answer

    def receive_pong(self, sock, ID, timeout):
        """
        Receive ping from the socket.
        """
        time_remaining = timeout
        while True:

```

```

start_time = time.time()
readable = select.select([sock], [], [], time_remaining)
time_spent = (time.time() - start_time)
if readable[0] == []: # Timeout
    return

time_received = time.time()
recv_packet, addr = sock.recvfrom(1024)
icmp_header = recv_packet[20:28]
type, code, checksum, packet_ID, sequence = struct.unpack(
    "bbHHh", icmp_header
)
if packet_ID == ID:
    bytes_In_double = struct.calcsize("d")
    time_sent = struct.unpack("d", recv_packet[28:28 + bytes_In_double])[
0]

    return time_received - time_sent

time_remaining = time_remaining - time_spent
if time_remaining <= 0:
    return

def send_ping(self, sock, ID):
    """
    Send ping to the target host
    """
    target_addr = socket.gethostbyname(self.target_host)

    my_checksum = 0

    # Create a dummy heder with a 0 checksum.
    header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, my_checksum, ID, 1)
    bytes_In_double = struct.calcsize("d")
    data = (192 - bytes_In_double) * "Q"
    data = struct.pack("d", time.time()) + bytes(data.encode('utf-8'))

    # Get the checksum on the data and the dummy header.
    my_checksum = self.do_checksum(header + data)
    header = struct.pack(
        "bbHHh", ICMP_ECHO_REQUEST, 0, socket.htons(my_checksum), ID, 1
    )
    packet = header + data
    sock.sendto(packet, (target_addr, 1))

def ping_once(self):
    """
    Returns the delay (in seconds) or none on timeout.
    """
    icmp = socket.getprotobyname("icmp")
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, icmp)

```

```

except socket.error as e:
    if e.errno == 1:
        # Not superuser, so operation not permitted
        print ("ICMP messages can only be sent from root user processes")
        sys.exit(2)
except Exception as e:
    print("Exception: %s" % (e))
    sys.exit(3)

my_ID = os.getpid() & 0xFFFF

self.send_ping(sock, my_ID)
delay = self.receive_pong(sock, my_ID, self.timeout)
sock.close()
return delay

def ping(self):
    """
    Run the ping process
    """
    for i in range(self.count):
        print("Ping to %s..." % self.target_host, )
        try:
            delay = self.ping_once()
        except socket.gaierror as e:
            print("Ping failed. (socket error: '%s')" % e[1])
            break

        if delay == None:
            print("Ping failed. (timeout within %ssec.)" % self.timeout)
        else:
            delay = delay * 1000
            print("Get pong in %0.4fms" % delay)

if __name__ == '__main__':
    pinger = Pinger(target_host=TARGET_HOST)
    pinger.ping()

```

Попробуйте запустить её и убедитесь, что программа отправляет 4 раза ICMP эхо-запрос и ожидает получения 4-х ICMP эхо-ответов.

```

(venv) ScrumBook:src ilya2$ sudo python ping-1.py
Ping to 8.8.8.8...
Get pong in 50.5660ms
Ping to 8.8.8.8...
Get pong in 32.7864ms
Ping to 8.8.8.8...
Get pong in 28.7151ms
Ping to 8.8.8.8...

```

```
Get pong in 37.4961ms
```

NOTE

Обратите внимание, что программа может работать только с правами суперпользователя (root). На MacOS я запускаю эту программу от пользователя root `sudo python ping-1.py`

Для реализации своей утилиты `ping` сделаем класс `class Pinger(object)`

```
class Pinger(object):

    def __init__(self, target_host, count=DEFAULT_COUNT, timeout=DEFAULT_TIMEOUT):
        self.target_host = target_host
        self.count = count
        self.timeout = timeout
```

В конструкторе определим переменные:

- `target_host` - кого будет пинговать.
- `count` - сколько раз отправлять ICMP эхо-запрос.
- `timeout` - время ожидания ICMP эхо-ответа.

Метод `ping` выполняет основную работу. Он в цикле вызывает функцию `ping_once` и печатает время, за которое пришел ответ. Если ответ не пришел, функция `ping` печатает "Ping failed".

```
def ping(self):
    """
    Run the ping process
    """
    for i in range(self.count):
        print("Ping to %s..." % self.target_host, )
        try:
            delay = self.ping_once()
        except socket.gaierror as e:
            print("Ping failed. (socket error: '%s')" % e[1])
            break

        if delay == None:
            print("Ping failed. (timeout within %ssec.)" % self.timeout)
        else:
            delay = delay * 1000
            print("Get pong in %0.4fms" % delay)
```

Самое интересное спрятано как раз внутри функции `ping_once`. Именно здесь мы создаем raw socket и работаем с ним.

```
def ping_once(self):
```

```

"""
Returns the delay (in seconds) or none on timeout.
"""

icmp = socket.getprotobyname("icmp")
try:
    sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, icmp)
except socket.error as e:
    if e.errno == 1:
        # Not superuser, so operation not permitted
        print("ICMP messages can only be sent from root user processes")
        sys.exit(2)
except Exception as e:
    print("Exception: %s" % (e))
    sys.exit(3)

my_ID = os.getpid() & 0xFFFF

self.send_ping(sock, my_ID)
delay = self.receive_pong(sock, my_ID, self.timeout)
sock.close()
return delay

```

При создании UDP сокета, мы указываем вторым параметром SOCK_DGRAM. При создании TCP сокета - SOCK_STREAM. Чтобы открыть raw socket, необходимо при создании сокета указать SOCK_RAW.

При создании raw socket необходимо указать протокол, который мы будем использовать. В нашем случае это ICMP. В IP заголовке есть поле [Protocol](#), оно указывает на следующий за IP заголовком протокол. Так как raw socket позволяет нам писать только данные в IP пакет, то мы должны как-то сообщить ОС, что мы собираемся самостоятельно сформировать именно ICMP пакет.

Когда ОС парсит IP заголовок, то для определения, какой протокол лежит внутри IP

Такой raw socket позволит нам самостоятельно сформировать заголовок ICMP и данные. И, чтобы IP протокол знал, ч