Лабораторная работа №6

Тема: «Сетевые приложения на Python»

Работа с сокетами

Применяемая в IP-сетях архитектура клиент-сервер использует IP-пакеты для коммуникации между клиентом и сервером. Клиент отправляет запрос серверу, на который тот отвечает. В случае с TCP/IP между клиентом и сервером устанавливается соединение (обычно с двусторонней передачей данных), а в случае с UDP/IP - клиент и сервер обмениваются пакетами (дейтаграммамми) с негарантированной доставкой.

Каждый сетевой интерфейс IP-сети имеет уникальный в этой сети адрес (IP-адрес). Упрощенно можно считать, что каждый компьютер в сети Интернет имеет собственный IP-адрес. При этом в рамках одного сетевого интерфейса может быть несколько сетевых портов. Для установления сетевого соединения приложение клиента должно выбрать свободный порт и установить соединение с серверным приложением, которое слушает (listen) порт с определенным номером на удаленном сетевом интерфейсе. Пара IP-адрес и порт характеризуют сокет (гнездо) – начальную (конечную) точку сетевой коммуникации. Для создания соединения TCP/IP необходимо два сокета: один на локальной машине, а другой – на удаленной. Таким образом, каждое сетевое соединение имеет IP-адрес и порт на локальной машине, а также IP-адрес и порт на удаленной машине.

Модуль ***socket*** обеспечивает возможность работать с сокетами из Python. Сокеты используют транспортный уровень согласно семиуровневой модели OSI (Open Systems Interconnection, взаимодействие открытых систем), то есть относятся к более низкому уровню, чем большинство описываемых в этом разделе протоколов.

Уровни модели OSI

Уровень	**Пояснение**	**Примеры**
Физический	Поток битов, передаваемых по физической линии. Определяет параметры физической линии.	
Канальный	Кодирует и декодирует данные в виде потока битов, справляясь с ошибками, возникающими на физическом уровне в пределах физически единой сети.	Ethernet, PPP, ATM

Уровень	**Пояснение**	**Примеры**
Сетевой	Маршрутизирует информационные пакеты от узла к узлу.	IP
Транспортный	Обеспечивает прозрачную передачу данных между двумя точками соединения.	TCP, UDP
Сеансовый	Управляет сеансом соединения между участниками сети. Начинает, координирует и завершает соединения.	ADSP, RPC, RTCP, SOCKS
Представления	Обеспечивает независимость данных от формы их представления путем преобразования форматов. На этом уровне может выполняться прозрачное (с точки зрения вышележащего уровня) шифрование и дешифрование данных.	XDR
Приложений	Поддерживает конкретные сетевые приложения. Протокол зависит от типа сервиса.	HTTP, FTP, SMTP, NNTP, POP3, IMAP

Каждый сокет относится к одному из коммуникационных доменов. Модуль ***socket*** поддерживает домены UNIX и Internet. Каждый домен подразумевает свое семейство протоколов и адресацию. Данное изложение будет затрагивать только домен Internet, а именно протоколы TCP/IP и UDP/IP, поэтому для указания коммуникационного домена при создании сокета будет указываться константа `socket.AF_INET`.

В качестве примера следует рассмотреть простейшую клиент-серверную пару. Сервер будет принимать строку и отвечать клиенту. Сетевое устройство иногда называют хостом (*host*), поэтому будет употребляться этот термин по отношению к компьютеру, на котором работает сетевое приложение.

Пример клиент-серверной программы:

Сервер

```
import socket

HOST = '127.0.0.1'
PORT = 50007
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

Клиент

```
import socket

HOST = '127.0.0.1'
PORT = 50007
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

Документация: https://docs.python.org/3/library/socket.html

Прежде всего, нужно запустить сервер, затем, запустив еще одну копию IDLE, запустить в ней клиент. Сервер открывает сокет на локальной машине на порту 50007, и адресе 127.0.0.1 (метод ***bind*** связывает локальный сетевой адрес транспортного уровня с сокетом). После этого сервер слушает (***listen***) порт. Когда на порту появляются данные, принимается (***accept***) входящее соединение, создается сокет, соответствующий новому соединению клиента и сервера. Сокет, для которого был вызван ***accept***, остается в состоянии ***listen*** и готов к принятию следующих соединений. Метод ***accept*** возвращает пару: *socket*-объект и адрес удаленного компьютера, устанавливающего соединение (пара: IP-адрес, порт на удаленной машине). После этого можно применять методы ***recv*** и ***send*** для общения с клиентом. В ***recv*** задается число байтов в очередной порции, от клиента может прийти и меньшее количество данных.

Модуль smtplib

Сообщения электронной почты в Интернете передаются от клиента к серверу и между серверами в основном по протоколу **SMTP** (Simple Mail Transfer Protocol, простой протокол передачи почты). Протокол **SMTP** и **ESMTP** (расширенный вариант SMTP) описаны в RFC 821 и RFC 1869. Для работы с SMTP в стандартной библиотеке модулей имеется модуль ***smtplib***. Для того чтобы начать SMTP-соединение с сервером электронной почты, необходимо сначала создать объект для управления SMTP-сессией с помощью конструктора класса SMTP:

`smtplib.SMTP([host[, port]])`

Параметры **host** и **port** задают адрес и порт SMTP-сервера, через который будет отправляться почта. По умолчанию, **port=25**. Если host задан, конструктор сам установит соединение, иначе придется отдельно вызывать метод `connect()`. Экземпляры класса SMTP имеют методы для всех распространенных команд SMTP-протокола, но для отправки почты достаточно вызова конструктора и методов `sendmail()` и `quit()`. Класс SMTP поддерживает оператор ***with***.

Пример отправки email сообщения:

```
from smtplib import SMTP

msg = 'Hello!'
username = 'example@gmail.com'
password = 'my_password'

with SMTP("smtp.gmail.com:587") as smtp:
    smtp.starttls()
    smtp.login(username, password)
    smtp.sendmail(
        'example@gmail.com',
        'mail_to@example.domain',
        msg
    )
```

Дополнительная информация:

https://support.google.com/mail/answer/7126229?hl=ru

https://support.google.com/accounts/answer/185833

Модуль imap

Еще один протокол – IMAP (Internet Message Access Protocol). IMAP предоставляет пользователю широкие возможности для работы с почтовыми ящиками, находящимися на почтовом сервере. Почтовая программа, использующая этот протокол, получает доступ к хранилищу корреспонденции на сервере так, как будто эта корреспонденция расположена на компьютере получателя.

Пример получения email сообщений:

```
from imaplib import IMAP4_SSL

with IMAP4_SSL("imap.gmail.com", 993) as M:
    rc, resp = M.login('example@gmail.com', 'my_password')
    M.select()
    typ, data = M.search(None, 'ALL')
    for num in data[0].split():
        typ, data = M.fetch(num, '(RFC822)')
        print('Message %s\n%s\n' % (num, data[0][1]))
```

Дополнительная информация:

https://support.google.com/mail/answer/7126229

Задание

Написать приложение **Mailer**

Структура приложения:

Файл	**Пояснение**
client.py	Скрипт, который принимает от пользователя email-адрес и текст сообщения (*ввод данных через клавиатуру*), отправляет данную информацию (*используя сокеты*) на сервер (*server.py*). Если сервер отвечает «OK», то скрипт завершает работу, иначе выводит пользователю информацию об ошибке и предлагает ввести данные заново.
server.py	Скрипт, который «слушает» сообщения от *client.py*. При получении *email пользователя* и текста сообщения проводит валидацию входных данных и в случае ошибки отправляет информацию об ошибке в *client.py*. Если данные корректны, то скрипт отправляет текст сообщения на email пользователя (*отправитель – email администратора*) и email администратора. Заголовок письма в формате: «[Ticket #12345] Mailer», где 12345 – уникальный ID, формируемый скриптом *server.py*. В случае ошибки возвращает *client.py* соответствующую информацию, иначе отправляет «OK».
success_request.log	Лог-файл, в котором хранится информация об успешно обработанных сообщениях от *collector.py*
error_request.log	Лог-файл, в котором хранится информация об ошибочных сообщениях от *collector.py*
collector.py	Скрипт, который с заданной частотой проверяет новые сообщения, поступающие на email администратора. Если тема письма имеет формат «[Ticket #12345] Mailer», то дописывает в файл *success_request.log* ID и текст полученного сообщения, иначе дописывает в файл *error_request.log* текст сообщения.

Файл	**Пояснение**

.env	Структурированный текстовый файл формата:
	`KEY_1=VALUE_1`
	`KEY_2=VALUE_2`
	
	В файле должна храниться следующая информация:
	1. Email логин администратора
	2. Email пароль администратора
	3. Хост почтового сервера (чтение сообщений)
	4. Порт почтового сервера (чтение сообщений)
	5. Хост почтового сервера (отправка сообщений)
	6. Порт почтового сервера (отправка сообщений)
	7. Частота проверки новых сообщений (в секундах)
	
	Пример:
	`EMAIL_LOGIN=example@gmail.com`
	`EMAIL_PASSWORD=my_password`
	`IMAP_HOST= imap.gmail.com`
	`IMAP_PORT=993`
	`SMTP_HOST= smtp.gmail.com`
	`SMTP_PORT=587`
	`PERIOD_CHECK=60`