**Простейшие TCP-клиент и эхо-сервер**

**Цель работы**

Познакомиться с приемами работы с сетевыми сокетами в языке программирования Python.

**Задания для выполнения**

1. Создать простой TCP-сервер, который принимает от клиента строку (порциями по 1 КБ) и возвращает ее. (Эхо-сервер).
2. Сервер должен выводить в консоль служебные сообщения (с пояснениями) при наступлении любых событий:
   1. Запуск сервера;
   2. Начало прослушивания порта;
   3. Подключение клиента;
   4. Прием данных от клиента;
   5. Отправка данных клиенту;
   6. Отключение клиента;
   7. Остановка сервера.
3. Напишите простой TCP-клиент, который устанавливает соединение с сервером, считывает строку со стандартного ввода и посылает его серверу.
4. Клиент должен выводить в консоль служебные сообщения (с пояснениями) при наступлении любых событий:
   1. Соединение с сервером;
   2. Разрыв соединения с сервером;
   3. Отправка данных серверу;
   4. Прием данных от сервера.

Изображение выглядит как текст

Автоматически созданное описание

**Методические указания**

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 (датаграммами) 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.

Прежде всего нам необходимо создать сокет:

sock = socket.socket()

Здесь ничего особенного нет и данная часть является общей и для клиентских и для серверных сокетов. Дальше мы будем писать код для сервера.

**Сервер**

Теперь нам нужно определится с хостом и портом для нашего сервера. Насчет хоста — мы оставим строку пустой, чтобы наш сервер был доступен для всех интерфейсов. А порт возьмем любой от нуля до 65535. Следует отметить, что в большинстве операционных систем прослушивание портов с номерами 0 — 1023 требует особых привилегий. Для примера выберем порт 9090. Теперь свяжем наш сокет с данными хостом и портом с помощью метода bind, которому передается кортеж, первый элемент (или нулевой, если считать от нуля) которого — хост, а второй — порт:

sock.bind(('', 9090))

Теперь у нас все готово, чтобы принимать соединения. С помощью метода listen мы запустим для данного сокета режим прослушивания. Метод принимает один аргумент — максимальное количество подключений в очереди. Установим его в единицу:

sock.listen(1)

Ну вот, наконец-то, мы можем принять подключение с помощью метода accept, который возвращает кортеж с двумя элементами: новый сокет и адрес клиента. Именно этот сокет и будет использоваться для приема и посылке клиенту данных.

conn, addr = sock.accept()

Вот и все. Теперь мы установили с клиентом связь и можем с ним «общаться». Т.к. мы не можем точно знать, что и в каких объемах клиент нам пошлет, то мы будем получать данные от него небольшими порциями. Чтобы получить данные нужно воспользоваться методом recv, который в качестве аргумента принимает количество байт для чтения. Мы будем читать порциями по 1024 байт (или 1 кб):

while True:

data = conn.recv(1024)

if not data:

break

conn.send(data.upper())

Как мы и говорили для общения с клиентом мы используем сокет, который получили в результате выполнения метода accept. Мы в бесконечном цикле принимаем 1024 байт данных с помощью метода recv. Если данных больше нет, то этот метод ничего не возвращает. Таким образом мы можем получать от клиента любое количество данных.

Дальше в нашем примере для наглядности мы что-то сделаем с полученными данными и отправим их обратно клиенту. Например, с помощью метода upper у строк вернем клиенту строку в верхнем регистре.

Теперь можно и закрыть соединение:

conn.close()

Собственно сервер готов. Он принимает соединение, принимает от клиента данные, возвращает их в виде строки в верхнем регистре и закрывает соединение.

**Клиент**

Думаю, что теперь будет легче. Да и само клиентское приложение проще — нам нужно создать сокет, подключиться к серверу послать ему данные, принять данные и закрыть соединение. Все это делается так:

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import socket

sock = socket.socket()

sock.connect(('localhost', 9090))

sock.send('hello, world!')

data = sock.recv(1024)

sock.close()

print data

Думаю, что все понятно, т.к. все уже разбиралось ранее. Единственное новое здесь — это метод connect, с помощью которого мы подключаемся к серверу. Дальше мы читаем 1024 байт данных и закрываем сокет.

**Контрольные вопросы**

1. Чем отличаются клиентские и серверные сокеты?
2. Как можно передавать через сокеты текстовую информацию?
3. Какие операции с сокетами блокируют выполнение программы?
4. Что такое неблокирующие сокеты?
5. В чем преимущества и недостатки использования TCP по сравнению с UDP?
6. Какие системные вызовы, связанные с сокетами используются только на стороне сервера?
7. На каком уровне модели OSI работают сокеты?

**Задания для самостоятельного выполнения**

1. Модифицируйте код сервера таким образом, чтобы при разрыве соединения клиентом он продолжал слушать данный порт и, таким образом, был доступен для повторного подключения.
2. Модифицируйте код клиента и сервера таким образом, чтобы номер порта и имя хоста (для клиента) они спрашивали у пользователя. Реализовать безопасный ввод данных и значения по умолчанию.
3. Модифицировать код сервера таким образом, чтобы все служебные сообщения выводились не в консоль, а в специальный лог-файл.
4. Модифицируйте код сервера таким образом, чтобы он автоматически изменял номер порта, если он уже занят. Сервер должен выводить в консоль номер порта, который он слушает.
5. Реализовать сервер идентификации. Сервер должен принимать соединения от клиента и проверять, известен ли ему уже этот клиент (по IP-адресу). Если известен, то поприветствовать его по имени. Если неизвестен, то запросить у пользователя имя и записать его в файл. Файл хранить в произвольном формате.
6. Реализовать сервер аутентификации. Похоже на предыдущее задание, но вместе с именем пользователя сервер отслеживает и проверяет пароли. Дополнительные баллы за безопасное хранение паролей. Дополнительные баллы за поддержание сессии на основе токена наподобие cookies
7. Напишите вспомогательные функции, которые реализуют отправку и принятие текстовых сообщений в сокет. Функция отправки должна дополнять сообщение заголовком фиксированной длины, в котором содержится информация о длине сообщения. Функция принятия должна читать сообщение с учетом заголовка. В дополнении реализуйте преобразование строки в байтовый массив и обратно в этих же функциях. Дополнително оценивается, если эти функции будут реализованы как унаследованное расширение класса socket библиотеки socket.
8. Дополните код клиента и сервера таким образом, чтобы они могли посылать друг другу множественные сообщения один в ответ на другое.
9. Напишите многопользовательский чат. Подсказка: используйте сокеты, основанные на протоколе UDP.