Урок 3

Основы сетевого программирования

Введение в сетевое взаимодействие, его протоколы. Сокеты как основа работы сетевых приложений. Протоколы обмена в курсовом проекте.

Введение в сетевое взаимодействие

Программные и аппаратные средства, информационные ресурсы и знания объединяются в общую инфраструктуру, где у участников сети есть возможность распределенного (сетевого) взаимодействия. Его обеспечивают клиент-серверные приложения, через которые доступны сетевые ресурсы.

Главные участники сетевого взаимодействия — клиентская и серверная стороны. Это любые компьютеры, объединенные локальной или глобальной сетью. Но сервер и клиент — это не обязательно обособленные компоненты оборудования. Они могут быть реализованы не физически, а программно. Выполнять функции клиента и сервера могут приложения, функционирующие на одном вычислительном устройстве. Клиентская сторона инициирует запрос на выполнение действий, сервер получает и выполняет его, формирует ответ и возвращает его клиенту.

На практике для реализации клиент-серверного взаимодействия, то есть разработки программных приложений, обеспечивающих связь клиента и сервера, используются высокоуровневые языки программирования — в том числе Python. В нем есть средства для взаимодействий между вычислительными устройствами, объединенными в сеть.

Важнейшими элементами сетевого взаимодействия являются сокеты (с англ. — «гнездо»). Это программные объекты (интерфейсы), определяющие конечную точку соединения. Сокет, описывающий параметры сетевого подключения, выполняет функцию файла. Программное приложение, обеспечивающее сетевое взаимодействие, может извлекать данные из сокета или записывать их в него. При этом весь процесс работы с сокетом аналогичен файловому взаимодействию.

Протоколы сетевого взаимодействия

Когда люди общаются, придерживаются правил, чтобы информация была понятной: говорят на одном языке, в ровном темпе, слушают собеседника. При сетевом взаимодействии компьютеров тоже есть определенные правила, чтобы одна машина могла корректно трактовать данные, переданные от другой.

Обмен данными между вычислительными устройствами осуществляется пакетами. Протокол определяет правила упаковки данных, передачи (например, скорость), распаковки в исходную форму.

При разработке программных приложений на базе Python, реализующих логику сетевого взаимодействия, важно разобраться с протоколами транспортного уровня: TCP и UDP и, в первую очередь, TCP. Именно он определяет правила выполнения большинства задач: подключения к базам данных, обеспечения сетевого взаимодействия, работы с веб-сервисами.

Основы разработки сетевых приложений

Стандартная библиотека Python поддерживает как операции с сокетами, так и функции для работы с прикладными протоколами высокого уровня, такими как HTTP. Кратко познакомимся с разработкой сетевых приложений. За дополнительной информацией рекомендуем обращаться к специализированным книгам, например «UNIX. Разработка сетевых приложений» Уильяма Ричарда Стивенса (W. Richard Stevens).

Сокеты

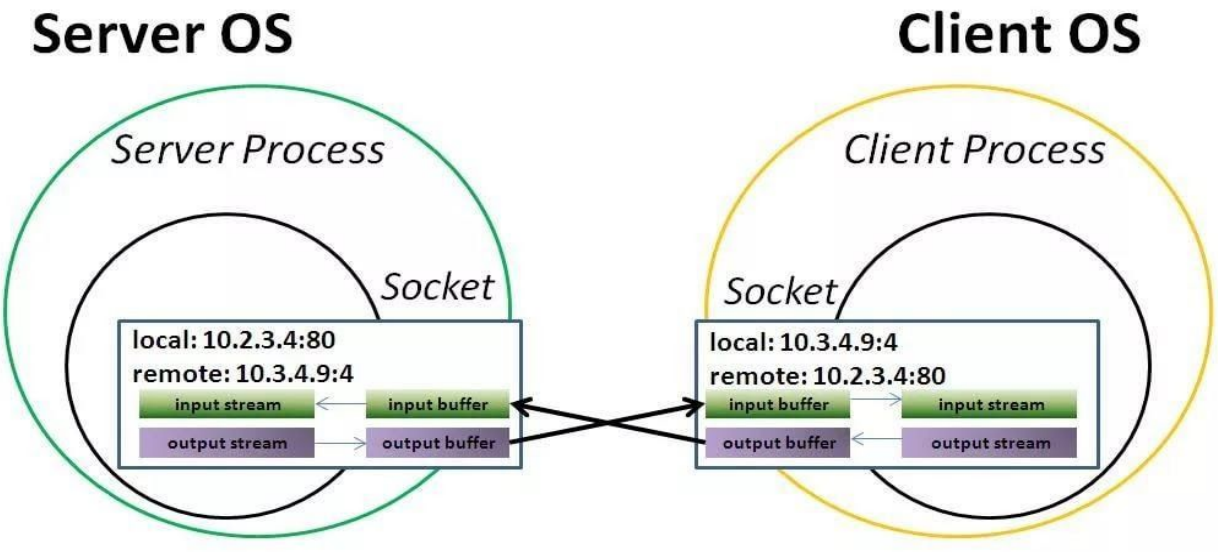
Модули, входящие в стандартную библиотеку Python и предназначенные для разработки сетевых приложений, главным образом поддерживают два интернет-протокола: TCP и UDP. Первый — надежный протокол с созданием логического соединения. TCP используют, чтобы формировать между компьютерами двусторонний канал обмена данными. Благодаря TCP пакеты гарантированно доставляются с соблюдением порядка их очередности, с автоматическим разбиением данных на пакеты и контролем их передачи. В то же время TCP работает медленно, так как потерянные пакеты многократно повторно отправляются, а операций, выполняемых над пакетами, слишком много.

Протокол UDP — низкоуровневый. С его помощью компьютеры могут отправлять и получать информацию в виде отдельных пакетов, не создавая логическое соединение. В отличие от TCP, взаимодействия по протоколу UDP не отличаются надежностью. Это усложняет управление ими в приложениях, в которых при обмене информацией нужны гарантии. Поэтому большинство интернет-приложений используют TCP.

Так что там, где необходима гарантированная передача данных без жесткой привязки к фактору времени (например, в веб-браузере, telnet, почтовом клиенте), применяется TCP-протокол. А для отправки данных в режиме реального времени (в многопользовательских игровых приложениях, для звука и видео) — UDP.

С обоими протоколами работают с помощью программной абстракции, известной как *сокет*. Это объект, напоминающий файл и позволяющий программе принимать входящие соединения, устанавливать исходящие, а также отправлять и принимать данные. Сокет — особая структура на уровне операционной системы. Ее описывают два параметра:

1. **IP-адрес —** идентификатор вычислительного устройства в сети.
2. **Порт** — число, уникальное для данного вычислительного устройства. В рамках операционной системы за определенным портом может быть закреплено только одно приложение.



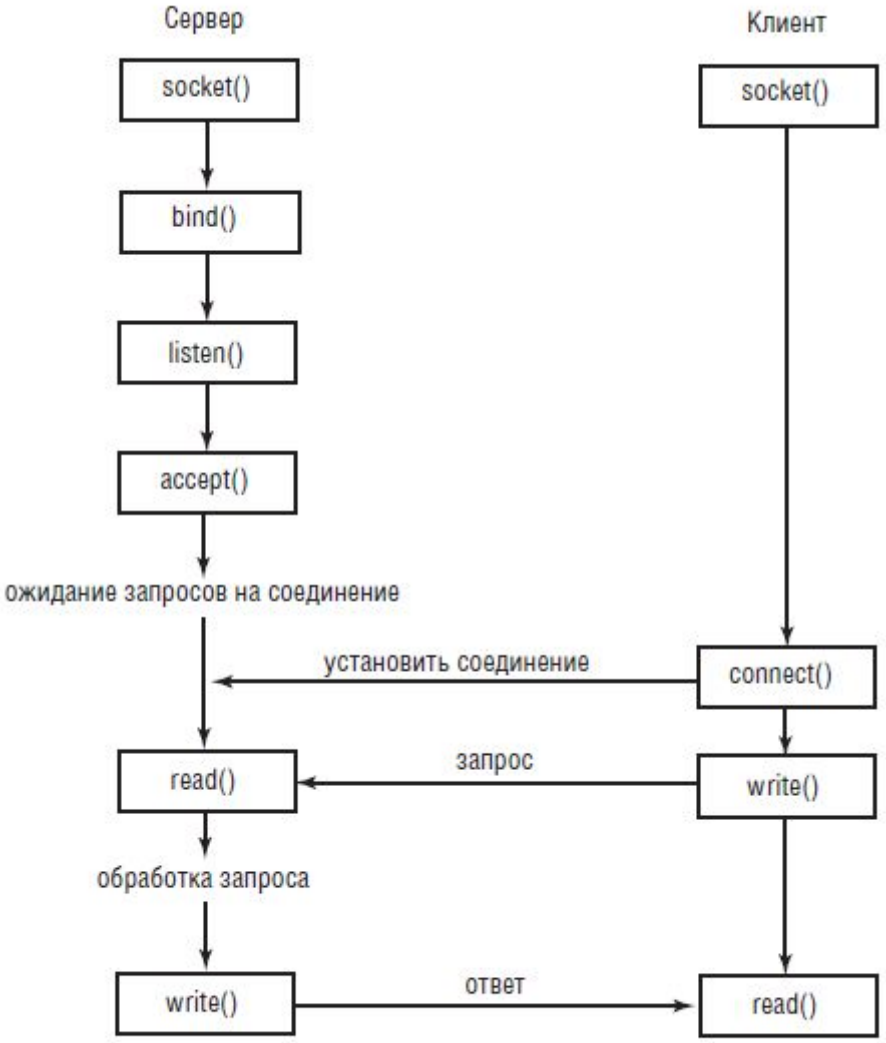
Компьютер, принимающий соединение (сервер), должен присвоить своему объекту сокета определенный номер порта. **Порт** — это 16-битное число в диапазоне 0–65535, которое используется клиентами для уникальной идентификации серверов. Порты с номерами 0–1023 зарезервированы для системы и используются наиболее распространенными сетевыми протоколами. Ниже перечислены некоторые из них с присвоенными номерами портов (более полный список можно найти здесь):

Служба	Номер порта
FTP-Data	20
FTP-Control	21
SSH	22
Telnet	23
SMTP (электронная почта)	25
HTTP (WWW)	80
IMAP	143
HTTPS (безопасный WWW)	443

Поскольку в клиент-серверном взаимодействии участвуют две стороны, для каждой должен быть создан объект сокета. Чтобы запустить сетевое взаимодействие, необходимо открыть сокет на одной стороне и соединить его с сокетом на другой. Так произойдет инициализация виртуального соединения, через которое могут передаваться данные.

Процесс, запущенный на стороне сервера, может создать (открыть) слушающий сокет и привязать его к порту операционной системы. Слушающий процесс переходит в режим ожидания и активизируется при появлении нового соединения. При этом можно проверить актуальные активные соединения, установить периодичность операции.

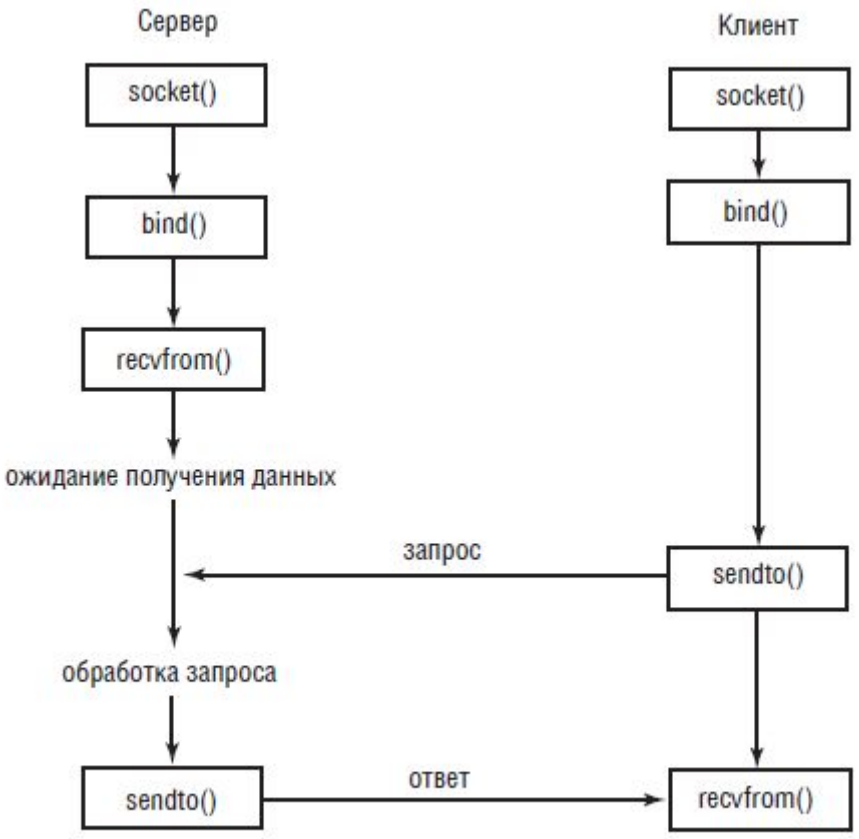
Процедура установки TCP-соединения между клиентом и сервером определяется точной последовательностью операций, показанной на рисунке:



Процедура установки TCP-соединения

На стороне сервера, работающего по протоколу TCP, есть объект сокета для приема запросов на соединение. Это **не тот же самый сокет**, что в дальнейшем используется для обмена данными с клиентом. Системный вызов **accept()** возвращает новый объект сокета, который фактически будет применяться для поддержки соединения. Это позволяет серверу одновременно обслуживать соединения с большим количеством клиентов.

Взаимодействия по протоколу UDP выполняются похожим способом, но клиенты и серверы не устанавливают логическое соединение друг с другом (см. рисунок):



Обмен данными по протоколу UDP

В Python для работы с сокетами применяется модуль **socket**, в котором реализованы функции, отвечающие за создание нового сокета, установление и закрытие соединения, отправку данных по сети и их получение и другое. В таблице приведен список и описание функций для сокетов на серверной и клиентской стороне, а также общих.

Общие	Серверные	Клиентские
socket — создать сокет	**bind** — привязать сокет к IP-адресу и порту машины	**connect** — установить соединение
send — передать данные	**listen** — просигнализировать о готовности принимать соединения	
recv — получить данные	**accept** — принять запрос на установку соединения	
close — закрыть соединение		

Рассмотрим, как протокол TCP применяется клиентом и сервером посредством модуля **socket**. В этом примере сервер просто возвращает клиенту текущее время в виде строки (файл **examples/01_sockets/time_server.py**):

```
# Программа сервера времени
from socket import *
import time

s = socket(AF_INET, SOCK_STREAM)   # Создает сокет TCP
s.bind(('', 8888))                 # Присваивает порт 8888
s.listen(5)                        # Переходит в режим ожидания запросов;
                                   # одновременно обслуживает не более
                                   # 5 запросов.
while True:
    client, addr = s.accept()      # Принять запрос на соединение
    print("Получен запрос на соединение от %s" % str(addr))
    timestr = time.ctime(time.time()) + "\n"
    client.send(timestr.encode('ascii'))
    client.close()
```

Вызов функции **socket()** запускает создание сокета. Основные параметры данной функции — это **communication domain** и **type of socket**. В качестве коммуникационного домена, как правило, передается значение **AF_INET**, — оно указывает, что создаваемый сокет будет сетевым. В качестве типа сокета указывается **SOCK_STREAM** — он определяет сокет как потоковый, то есть реализующий последовательный, надежный двусторонний поток байтов. В результате функции **socket()** создается конечная точка соединения и возвращается файловый дескриптор, который позволяет работать с сокетом, как с файлом — записывать и считывать данные в/из него. Таким образом, константа **SOCK_STREAM** указывает на то, что сокет работает с TCP-пакетами — то есть это TCP-пакет.

Клиентская программа (файл **examples/01_sockets/client_server.py**):

```
# Программа клиента, запрашивающего текущее время
from socket import *

s = socket(AF_INET,SOCK_STREAM)   # Создать сокет TCP
s.connect(('localhost', 8888))    # Соединиться с сервером
tm = s.recv(1024)                 # Принять не более 1024 байтов данных
s.close()
print("Текущее время: %s" % tm.decode('ascii'))
```

В сетевых протоколах обмен данными должен выполняться в байтовом формате. Поэтому надо кодировать строки, передаваемые через сеть. Именно по этой причине в программе сервера к отправляемым данным применяется метод **encode('ascii')**. Они поступают в виде простой последовательности кодированных байтов: как и когда клиент принимает данные из сети. Если вывести эту последовательность на экран или попробовать интерпретировать ее как текст — результат, скорее всего, получится совсем не тем, какого вы ожидали. Поэтому прежде чем работать с данными, их необходимо декодировать. Для этого в программе клиента к принимаемым данным применяется метод **decode('ascii')**. Обратите внимание на файл с примером кода для кодирования строк и байтов — **examples/01_sockets/str_bytes_bytearray.py**).

```
# В Python 3 все строки - строки юникода
s = 'Python'
```

```
# Отдельный тип - строка байтов
bs = b'Python'

# Отдельный тип - bytearray - изменяемая строка байтов
ba = bytearray(bs)

# Преобразования между строками
s2 = bs.decode('cp1251')          # Из байт-строки в юникод строку
bs2 = s.encode('koi8-r')          # Из юникод-строки в строку байтов
ba2 = bytearray(s, 'utf-8')       # Из юникод-строки в массив байтов
```

Рассмотрим еще один пример, демонстрирующий возможности сокетов. Представленная связка серверного и клиентского сокетов реализуют вариант простейшего сетевого взаимодействия: клиент запрашивает соединение с сервером и отправляет ему строковые данные. Сервер получает это строковое сообщение и возвращает его обратно на клиентскую сторону.

Серверная программа (файл **examples/01_sockets/data_server.py**):

```
# Программа сервера для получения приветствия от клиента и отправки ответа
from socket import *
import time

s = socket(AF_INET, SOCK_STREAM)  # Создает сокет TCP
s.bind(('', 8007))                # Присваивает порт 8888
s.listen(5)                       # Переходит в режим ожидания запросов;
                                  # Одновременно обслуживает не более
                                  # 5 запросов.
while True:
    client, addr = s.accept()
    data = client.recv(1000000)
    print('Сообщение: ', data.decode('utf-8'), ', было отправлено клиентом: ',
addr)
    msg = 'Привет, клиент'
    client.send(msg.encode('utf-8'))
    client.close()
```

Клиентская программа (файл **examples/01_sockets/data_client.py**):

```
# Программа клиента для отправки приветствия серверу и получения ответа
from socket import *

s = socket(AF_INET, SOCK_STREAM)  # Создать сокет TCP
s.connect(('localhost', 8007))    # Соединиться с сервером
msg = 'Привет, сервер'
s.send(msg.encode('utf-8'))
data = s.recv(1000000)
print('Сообщение от сервера: ', data.decode('utf-8'), ', длиной ', len(data), '
байт')
s.close()
```

Синтаксис сокетов на базе UDP-протокола почти идентичен — за исключением нескольких моментов. В частности, для параметра **type of socket** необходимо определить значение **SOCK_DGRAM**, которое показывает, что создается UDP-сокет. Также для сокета в данном случае определяются параметры **SO_REUSEADDR** и **SO_BROADCAST**, открывающие доступ сокету к нескольким приложениям и широковещательным пакетам. Приведенная ниже серверная программа обеспечивает вывод сообщений от клиента при каждом его запросе на соединение (файл **examples/01_sockets/udp_server.py**):

```
# Программа вывода сообщений на стороне сервера при запросе от клиента
from socket import *

s = socket(AF_INET, SOCK_DGRAM)            # Определяем UDP-протокол
s.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1) # Несколько приложений может слушать
сокет
s.setsockopt(SOL_SOCKET, SO_BROADCAST, 1) # Определяем широковещательные пакеты
s.bind(('', 8888))
while True:
      msg = s.recv(128)
      print(msg)
```

Клиентская программа (файл **examples/01_sockets/udp_client.py**):

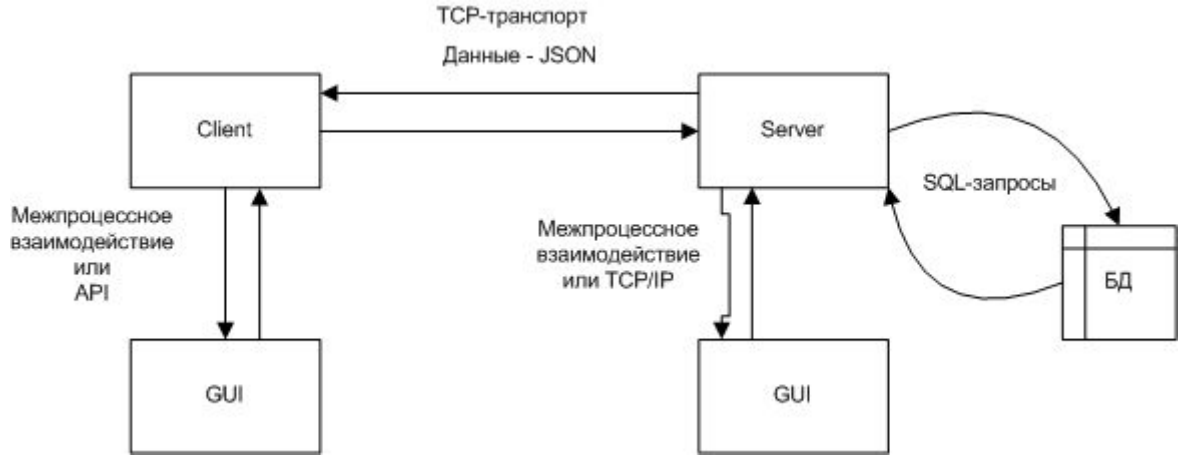
```
# Программа клиента, передающего серверу сообщения при каждом запросе на
соединение
from socket import *

s = socket(AF_INET, SOCK_DGRAM)
s.setsockopt(SOL_SOCKET, SO_BROADCAST, 1)
while True:
      s.sendto('Запрос на соединение!',('',8888))
```

Протоколы обмена в курсовом проекте

На первом уроке мы обговорили, что домашними заданиями будет постепенная разработка цельного проекта. Предлагаем реализовать упрощенный аналог существующих мессенджеров. Проработаем как серверную, так и клиентскую часть.

Обобщенная схема взаимодействия элементов системы представлена на рисунке:



Протокол обмена

За основу протокола обмена между клиентом и сервером взят проект JIM (JSON instant messaging)

Протокол JIM базируется на передаче JSON-объектов через TCP-сокеты.

Все сетевые взаимодействия осуществляются в байтах.

Спецификация объектов

JSON-данные, пересылаемые между клиентом и сервером, обязательно должны содержать поля **action** и **time**.

Поле **action** задает тип сообщения между клиентом и сервером. А **time** —временная метка отправки JSON-объекта, UNIX-время (число секунд от 1 января 1970 года).

Например, для аутентификации надо сформировать JSON-объект:

```
{
 "action": "authenticate",
 "time": <unix timestamp>,
        "user": {
                "account_name": "C0deMaver1ck",
                "password":     "CorrectHorseBatterStaple"
        }
}
```

Ответы сервера должны содержать поле **response**, и могут — поле **alert/error** с текстом ошибки.

```
{
    "response": <код ответа>,
    "alert": <текст ответа>
}
```

Все объекты имеют ограничения длины (количество символов):

- поле **action** — 15 символов (сейчас самое длинное название — authenticate (11 символов); вряд ли должно понадобиться что-то больше);
- поле **response** — с кодом ответа сервера, это 3 цифры;
- **имя пользователя** / **название чата** (name): 25 символов;
- **сообщение** — максимум 500 символов (" ").

Итоговое ограничение для JSON-объекта — 640 символов (можно добавить дополнительные поля или изменить имеющиеся). Исходные ограничения на длину сообщений упрощают реализации клиента и сервера.

Подключение, отключение, авторизация

JIM-протокол не подразумевает обязательной авторизации при подключении к серверу. Это позволяет реализовать функционал для гостевых пользователей.

Если какое-то действие требует авторизации, сервер должен ответить соответствующим кодом ошибки — 401.

После подключения при необходимости авторизации клиент должен отправить сообщение с логином/паролем, например:

```
{
        "action": "authenticate",
        "time": <unix timestamp>,
        "user": {
                "account_name":  "C0deMaver1ck",
                "password":      "CorrectHorseBatteryStaple"
        }
}
```

В ответ сервер может прислать один из кодов:

```
{
    "response": 200,
    "alert":"Необязательное сообщение/уведомление"
}

{
    "response": 402,
    "error": "This could be "wrong password" or "no account with that name""
}

{
    "response": 409,
    "error": "Someone is already connected with the given user name"
}
```

Отключение от сервера должно сопровождаться сообщением **“quit”**:

```
{
    "action": "quit"
}
```

Присутствие

Каждый пользователь при подключении к серверу отсылает сервисное сообщение о присутствии — **presence** с необязательным полем **type**:

```
{
        "action": "presence",
        "time": <unix timestamp>,
        "type": "status",
        "user": {
                "account_name":  "C0deMaver1ck",
                "status":      "Yep, I am here!"
        }
}
```

Чтобы проверить доступность пользователя online, сервер выполняет probe-запрос:

```
{
        "action": "probe",
        "time": <unix timestamp>,
}
```

Probe-запрос может отправлять только сервер, проверяя доступность клиентов из контакт-листа. На probe-запрос клиент должен ответить простым presence-сообщением.

Коды ответов сервера

JIM-протокол использует коды ошибок HTTP. Перечислим поддерживаемые:

- **1xx** — информационные сообщения:
    - **100** — базовое уведомление;
    - **101** — важное уведомление.
- **2xx** — успешное завершение:
    - **200** — OK;
    - **201** (created) — объект создан;
    - **202** (accepted) — подтверждение.
- **4xx** — ошибка на стороне клиента:
    - **400** — неправильный запрос/JSON-объект;
    - **401** — не авторизован;
    - **402** — неправильный логин/пароль;
    - **403** (forbidden) — пользователь заблокирован;
    - **404** (not found) — пользователь/чат отсутствует на сервере;
    - **409** (conflict) — уже имеется подключение с указанным логином;
    - **410** (gone) — адресат существует, но недоступен (offline).
- **5xx** — ошибка на стороне сервера:
    - **500** — ошибка сервера.

Коды ошибок могут быть дополнены новыми.

Сообщения-ответы имеют следующий формат (в зависимости от кода ответа):

```
{
    "response": 1xx / 2xx,
    "time": <unix timestamp>,
    "alert": "message (optional for 2xx codes)"
}
```

Или такой:

```
{
    "response": 4xx / 5xx,
    "time": <unix timestamp>,
    "error": "error message (optional)"
}
```

Сообщение «Пользователь–Пользователь»

Простое сообщение имеет формат:

```
{
    "action": "msg",
    "time": <unix timestamp>,
    "to": "account_name",
    "from": "account_name",
    "encoding": "ascii",
    "message": "message"
}
```

Когда сервер видит действие **msg**, ему не нужно читать или парсить все сообщение — только проверить адресата и передать месседж ему.

Если поле **to** (адресат) имеет префикс **#** — это сообщение для группы. Обрабатывается, как приватное сообщение, но по шаблону «Пользователь-Чат»

В ответ на такое событие клиенту возвращается код ошибки.

Поле **encoding** указывает кодировку сообщения. Если нет, считается **ascii**.

Сообщение «Пользователь-Чат».

Обрабатывается, как и «Пользователь-Пользователь», но с дополнением:

- Имя чата имеет префикс **#** (то есть сервер должен проверять поле **to** для всех сообщений и переправлять месседж всем online-пользователям данного чата).

Сообщение:

```
{
    "action": "msg",
    "time": <unix timestamp>,
    "to": "#room_name",
    "from": "account_name",
    "message": "Hello World"
}
```

Присоединиться к чату:

```
{
    "action": "join",
    "time": <unix timestamp>,
    "room": "#room_name"
}
```

Покинуть чат:

```
{
    "action": "leave",
    "time": <unix timestamp>,
    "room": "#room_name"
}
```

Методы протокола (actions)

- **“action”: “presence”** — присутствие. Сервисное сообщение для извещения сервера о присутствии клиента online;
- **“action”: “probe”** — проверка присутствия. Сервисное сообщение от сервера для проверки присутствии клиента online;
- **“action”: “msg”** — простое сообщение пользователю или в чат;
- **“action”: “quit”** — отключение от сервера;
- **“action”: “authenticate”** — авторизация на сервере;
- **“action”: “join”** — присоединиться к чату;
- **“action”: “leave”** — покинуть чат.

Протокол может быть расширен новыми методами.

Практическое задание

1. Реализовать простое клиент-серверное взаимодействие по протоколу JIM (JSON instant messaging):
    a. клиент отправляет запрос серверу;

b. сервер отвечает соответствующим кодом результата.

Клиент и сервер должны быть реализованы в виде отдельных скриптов, содержащих соответствующие функции.

Функции клиента:

- сформировать presence-сообщение;
- отправить сообщение серверу;
- получить ответ сервера;
- разобрать сообщение сервера;
- параметры командной строки скрипта **client.py <addr> [<port>]**:
    - **addr** — ip-адрес сервера;
    - **port** — tcp-порт на сервере, по умолчанию 7777.

Функции сервера:

- принимает сообщение клиента;
- формирует ответ клиенту;
- отправляет ответ клиенту;
- имеет параметры командной строки:
    - **-p <port>** — TCP-порт для работы (по умолчанию использует 7777);
    - **-a <addr>** — IP-адрес для прослушивания (по умолчанию слушает все доступные адреса).

Дополнительные материалы

1. Программа-мечта начинающего питоновода.
2. Сетевое программирование. Учебное пособие.
3. Python. Сетевое программирование.
4. Программирование сокетов.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. David Beazley, Brian K. Jones. Python Cookbook. Third Edition (каталог «Дополнительные материалы»).
2. Дэвид Бизли. Python. Подробный справочник (каталог «Дополнительные материалы»).

3. Лучано Ромальо. Python. К вершинам мастерства (каталог «Дополнительные материалы»).
4. Клиент-сервер. Сетевое программирование.
5. Сетевое взаимодействие.
6. Сокеты. Сетевое программирование.
7. Python. Сетевые приложения на Python.
8. UNIX: разработка сетевых приложений.
9. Программирование на Python. Сетевое программирование.