**WebSocket**

WebSocket — протокол полнодуплексной двунаправленной связи поверх TCP соединения, предназначенный для обмена сообщениями между браузером и веб-сервером в режиме реального времени. Данный протокол расположен на 7 (прикладном) уровне модели OSI.

Протокол WebSocket обеспечивает взаимодействие между браузером и веб-сервером с более низкими накладными расходами, что облегчает передачу данных в режиме реального времени с сервера и на сервер. Это стало возможным благодаря стандартизированному способу отправки сервером контента в браузер без запроса со стороны клиента и обеспечения возможности передачи сообщений туда и обратно при сохранении соединения. Таким образом, между браузером и сервером может происходить двусторонняя передача данных.

**Отличия протокола WebSocket от протокола HTTP**

Один из недостатков протокола HTTP – это проблема постоянного соединения с сервером. Реализация HTTP протокола не предполагала подобного рода взаимодействия. Например, если мы хотим получить данные с сервера в браузер, то нужно сделать очередной запрос к серверу, и это предполагает перезагрузку страницы. То есть, если мы открыли сайт в браузере, загрузили страницу, просмотрели, и к этому времени на сервере изменилась данная страница, то нам нужно перезагрузить страницу, чтобы получить изменение.

Есть довольно большое количество задач, где нам нужно получить возможность общаться с сервером в режиме реального времени, используя HTTP-протокол. То есть, если на сервере будут изменения, то нужно получить эти изменение в браузере, без перезагрузки. Один из таких примеров – это чат, где люди общаются, и когда один другому отправляет сообщения, то сообщения видно получателю моментально, без перезагрузки страницы. Раньше, для создания такого вида приложения прибегали к различным ухищрениям, которые имитировали push-действия сервера. Один из таких примеров – это организованные на клиенте фреймы, которые перезагружаются раз в секунду и отправляют запросы на сервер.

В этом подходе имеются минусы — создается очень большое количество запросов на сервер и тяжело организовать правильную структуру приложений. А самая главная проблема – это то, что мы делаем эмуляцию реакций на серверное событие. Всегда клиент (браузер) получает данные с большой задержкой.

В рамках технологии WebSocket браузер и сервер в любой момент могут отправлять и принимать данные, то есть они становится равными участниками.

WebSocket устанавливает одно единственное соединение клиента с сервером. Для работы с WebSockets обе стороны (клиент и сервер) должны поддерживать данную технологию.

Протокол WebSocket поддерживается в большинстве основных браузеров, включая [Google Chrome](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=ru&u=https://en.wikipedia.org/wiki/Google_Chrome&usg=ALkJrhiI_aJ0aV7BJu20oAoTH6aMkCqq_w) , [Microsoft Edge](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=ru&u=https://en.wikipedia.org/wiki/Microsoft_Edge&usg=ALkJrhj2Fu3UUMWHArbOWuku-kLCGvAiFw) , [Internet Explorer](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=ru&u=https://en.wikipedia.org/wiki/Internet_Explorer&usg=ALkJrhiwe0k0MRrZi0MK_QjZFdGfG097mQ) , [Firefox](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=ru&u=https://en.wikipedia.org/wiki/Firefox&usg=ALkJrhgIuZymTnDy0GjrbQSLvbyUIWlIyg), [Safari](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=ru&u=https://en.wikipedia.org/wiki/Safari_(web_browser)&usg=ALkJrhge6FOTdZM-a9GBViBIxfjm-J2nzA) и [Opera](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=ru&u=https://en.wikipedia.org/wiki/Opera_web_browser&usg=ALkJrhhf9axH6Zk7aU5s00IRkzaTMlrvdg) .

Для обмена информацией между клиентом и сервером, клиент отправляет запрос на подтверждение WebSocket, для которого сервер возвращает ответ на установление связи с помощью WebSocket.

В запросе клиента каждая строка заканчивается на \r\n и в конце запроса должна быть дополнительная пустая строка.

Существует несколько различных версий протокола WebSocket: 75, 76, 07, RFC 6455.

**Описание версий протокола WebSocket**

В версии протокола 75 запрос клиента на установление соединения выглядел так:  
GET /demo HTTP/1.1  
Upgrade: WebSocket  
Connection: Upgrade  
Host: example.com  
Origin: http://example.com

WebSocket-Protocol: sample

Ответ сервера выглядел вот так:

HTTP/1.1 101 Web Socket Protocol Handshake

Upgrade: WebSocket

Connection: Upgrade

WebSocket-Origin: http://example.com

WebSocket-Location: ws://example.com/demo

WebSocket-Protocol: sample

После отправки сервером ответа клиенту соединение считается установленным, клиент и сервер могут обмениваться сообщениями по установленному соединению. Для отправки текстового сообщения необходимо сначала отправить нулевой байт, а уже после само сообщение.

2 июня 2010 года вышла новая версия протокола ­– 76. В этой версии была реализована защита от поддельных запросов. Данная версия протокола не является обратно совместимой.

В данной версии протокола, клиент, который хочет установить соединение, посылает серверу следующий запрос:

GET /demo HTTP/1.1

Upgrade: WebSocket

Connection: Upgrade

Sec-WebSocket-Key2: 4 @1 46546xW%0l 1 5

Host: example.com

Sec-WebSocket-Key1: 12998 5 Y3 1 .P00

Origin: http://example.com

WebSocket-Protocol: sample

^n:ds[4U

В клиентский запрос были добавлены заголовки «Sec-WebSocket-Key1», «Sec-WebSocket-Key2» и 8-байтовое тело запроса, которые генерируются клиентом случайным образом.

Ответ, отправляемый сервером на данный запрос:

HTTP/1.1 101 Web Socket Protocol Handshake

Upgrade: WebSocket

Connection: Upgrade

Sec-WebSocket-Origin: http://example.com

Sec-WebSocket-Location: ws://example.com/demo

Sec-WebSocket-Protocol: sample

8jKS'y:G\*Co,Wxa-

В ответе сервера содержатся новые названия заголовков «Sec-WebSocket-Origin», «Sec-WebSocket-Location», «Sec-WebSocket-Protocol» вместо «WebSocket-Origin», «WebSocket-Location», «WebSocket-Protocol» и 16-байтное тело ответа, вычисляемое следующим образом:

1. из строки со значением заголовка запроса Sec-WebSocket-Key1 исключить все символы, не являющиеся цифрами (не попадающие в диапазон '0'..'9');
2. полученную строку превратить в 64-битное целое число (для примера выше получим 1299853100);
3. полученное число разделить целочисленным делением на количество пробелов в исходной строке со значением заголовка;
4. полученное число представить в виде 4-байтового 32-битного числа в формате [big endian](https://ru.wikipedia.org/wiki/Big_endian): старший байт хранится по нулевому смещению;
5. проделать то же самое с заголовком Sec-WebSocket-Key2;
6. интерпретируя числа из пунктов 4) и 5) как 4-байтовые строки, конкатенировать их (сложить в одну строку) и добавить как строку тело запроса;
7. вычислить от полученной 16-байтной строки значение [MD5](https://ru.wikipedia.org/wiki/MD5) и записать это значение в тело ответа «как есть», без преобразования в какое-либо представление;

22 апреля 2011 года вышла версия протокола 07. Эта версия протокола  
отличается от версии 76 тем, что каждый байт передаваемых данных от клиента к серверу обязательно маскируется 4-байтовой маской. Она создается для каждого сообщения заново.

В передаваемом сообщении теперь имеется заголовок, в котором содержатся следующие данные:

фрагментировано ли сообщение;

* тип передаваемых данных;
* подвергалось ли сообщение маскировке;
* размер данных;
* маска;
* другие управляющие данные (ping, pong…).

В данной версии протокола запрос клиента выглядит следующим образом:  
GET /chat HTTP/1.1

Host: server.example.com

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==

Sec-WebSocket-Origin: http://example.com

Sec-WebSocket-Protocol: chat, superchat

Sec-WebSocket-Version: 7

Ответ сервера:

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=

Sec-WebSocket-Protocol: chat

Ответ содержит заголовок Sec-WebSocket-Protocol с единственным протоколом, выбраным сервером (chat) из всех поддерживаемых клиентом (chat, superchat).

Заголовок Sec-WebSocket-Accept формируется следующим образом:

1. взять строковое значение из заголовка Sec-WebSocket-Key и объединить со строкой 258EAFA5-E914-47DA-95CA-C5AB0DC85B11 (в приведённом примере получится dGhlIHNhbXBsZSBub25jZQ==258EAFA5-E914-47DA-95CA-C5AB0DC85B11)
2. вычислить бинарный хеш SHA-1 (бинарная строка из 20 символов) от полученной в первом пункте строки
3. закодировать хеш в [Base64](https://ru.wikipedia.org/wiki/Base64) (s3pPLMBiTxaQ9kYGzzhZRbK+xOo=)

11 декабря 2011 года протокол 07 приобрёл статус рабочего предложения (RFC), но теперь вместо заголовка Sec-WebSocket-Origin стал использоваться заголовок Origin, данная версия протокола получила название RFC 6455.

**Пример реализации простого echo сервера и клиента**

**WebSocket на Python**

**Сервер:**

import socket

import struct

import array

from hashlib import sha1

from base64 import b64encode

def unpack\_frame(data):

frame = {}

byte1, byte2 = struct.unpack\_from('!BB', data)

frame['fin'] = (byte1 >> 7) & 1

frame['opcode'] = byte1 & 0xf

masked = (byte2 >> 7) & 1

frame['masked'] = masked

mask\_offset = 4 if masked else 0

payload\_hint = byte2 & 0x7f

if payload\_hint < 126:

payload\_offset = 2

payload\_length = payload\_hint

elif payload\_hint == 126:

payload\_offset = 4

payload\_length = struct.unpack\_from('!H',data,2)[0]

elif payload\_hint == 127:

payload\_offset = 8

payload\_length = struct.unpack\_from('!Q',data,2)[0]

frame['length'] = payload\_length

payload = array.array('B')

payload.fromstring(data[payload\_offset + mask\_offset:])

if masked:

mask\_bytes = struct.unpack\_from('!BBBB',data,payload\_offset)

for i in range(len(payload)):

payload[i] ^= mask\_bytes[i % 4]

frame['payload'] = payload.tostring()

return frame

def pack\_frame(buf, opcode, base64=False):

if base64:

buf = b64encode(buf)

b1 = 0x80 | (opcode & 0x0f) # FIN + opcode

payload\_len = len(buf)

if payload\_len <= 125:

header = struct.pack('>BB', b1, payload\_len)

elif payload\_len > 125 and payload\_len < 65536:

header = struct.pack('>BBH', b1, 126, payload\_len)

elif payload\_len >= 65536:

header = struct.pack('>BBQ', b1, 127, payload\_len)

return header + buf

#создаём сокет

sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

#привязываем сокет к адресу и порту в соответствии с заданием

sock.bind(('',8000))

#устанавливаем сокет в режим прослушивания, количество подключений в очереди - 1

sock.listen(1)

while True:

#получаем сокет и адрес клиента

conn, addr = sock.accept()

data = conn.recv(1024)

#если клиент отключился, начинаем цикл ожидания соединения сначала

if not data:

continue

print(data.decode())

#получаем заголовки от клиента

headers = data.decode("utf-8").split("\r\n")

key = ""

for line in headers: #Итерируемся по строкам

parts = line.partition(": ") # Делим по ':'

if parts[0] == "Sec-WebSocket-Key":

key = parts[2] # Находим необходимый ключ

key += "258EAFA5-E914-47DA-95CA-C5AB0DC85B11"

Acckey=b64encode((sha1(key.encode())).digest())

resp = "HTTP/1.1 101 Switching Protocols\r\nUpgrade: websocket\r\nConnection: Upgrade\r\nSec-WebSocket-Accept: "

resp += Acckey.decode() + "\r\n\r\n"

print(resp)

conn.send(resp.encode())

while True:

data = conn.recv(1024)

print("Запакованные данные от клиента:")

print(data)

print("Распакованные данные от клиента:")

print(unpack\_frame(data)['payload'])

buf=pack\_frame(unpack\_frame(data)['payload'], 0x1)

print("Запакованные данные для клиента:")

print(buf)

conn.send(buf)

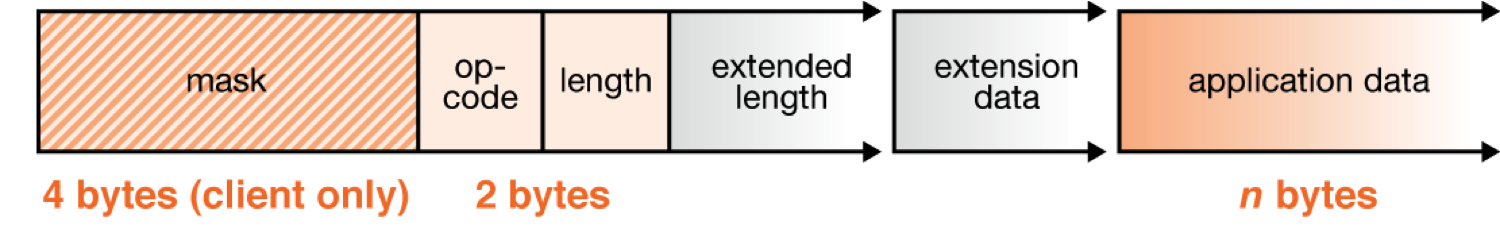
if not unpack\_frame(data)['payload']:

break

#закрываем соединение с клиентом

conn.close()

sock.close()

Пересылаемые кадры имеют вид:

Метод pack\_frame формирует кадры, а метод unpack\_frame извлекает исходные данные из кадра.

**Клиент:**

import socket

import struct #из него нам нужна функция pack() и unpack\_from()

import array # функция array()

from hashlib import sha1 #Кодирование Access Key о котором будет дальше

from base64 import b64encode #Кодирование Access Key о котором будет дальше

def unpack\_frame(data):

frame = {}

byte1, byte2 = struct.unpack\_from('!BB', data)

frame['fin'] = (byte1 >> 7) & 1

frame['opcode'] = byte1 & 0xf

masked = (byte2 >> 7) & 1

frame['masked'] = masked

mask\_offset = 4 if masked else 0

payload\_hint = byte2 & 0x7f

if payload\_hint < 126:

payload\_offset = 2

payload\_length = payload\_hint

elif payload\_hint == 126:

payload\_offset = 4

payload\_length = struct.unpack\_from('!H',data,2)[0]

elif payload\_hint == 127:

payload\_offset = 8

payload\_length = struct.unpack\_from('!Q',data,2)[0]

frame['length'] = payload\_length

payload = array.array('B')

payload.fromstring(data[payload\_offset + mask\_offset:])

if masked:

mask\_bytes = struct.unpack\_from('!BBBB',data,payload\_offset)

for i in range(len(payload)):

payload[i] ^= mask\_bytes[i % 4]

frame['payload'] = payload.tostring()

return frame

def pack\_frame(buf, opcode, base64=False):

if base64:

buf = b64encode(buf)

b1 = 0x80 | (opcode & 0x0f) # FIN + opcode

payload\_len = len(buf)

if payload\_len <= 125:

header = struct.pack('>BB', b1, payload\_len)

elif payload\_len > 125 and payload\_len < 65536:

header = struct.pack('>BBH', b1, 126, payload\_len)

elif payload\_len >= 65536:

header = struct.pack('>BBQ', b1, 127, payload\_len)

return header + buf

#создаём сокет

sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

#создаём запрос

request = b"GET /chat HTTP/1.1\r\nHost: localhost\r\nUpgrade: websocket\r\nConnection: Upgrade\r\nSec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==\r\nSec-WebSocket-Origin: localhost\r\nSec-WebSocket-Protocol: chat, superchat\r\nSec-WebSocket-Version: 7\r\n\r\n"

sock.connect(('localhost', 8000))

#через сокет отправляем запрос серверу

sock.send(request)

#получаем ответ сервера и выводим на консоль

resp = sock.recv(1024)

headers = resp.decode("utf-8").split("\r\n")

key = ""

for line in headers: #Итерируемся по строкам

parts = line.partition(": ") # Делим по ':'

if parts[0] == "Sec-WebSocket-Accept":

key = parts[2] # Находим необходимый ключ

validate = "dGhlIHNhbXBsZSBub25jZQ==258EAFA5-E914-47DA-95CA-C5AB0DC85B11"

Acckey=b64encode((sha1(validate.encode())).digest()).decode()

print(resp)

if Acckey == key:

print("Соединение установлено")

while True:

print("Введите сообщение:")

message=input()

print("Отправка сообщения серверу")

print("Сообщение:")

print(message)

buf = pack\_frame(message.encode(), 0x1)

print("Сообщение в запакованном виде:")

print(buf)

sock.send(buf)

print("Ответ сервера в запакованном виде:")

resp = sock.recv(1024)

print(resp)

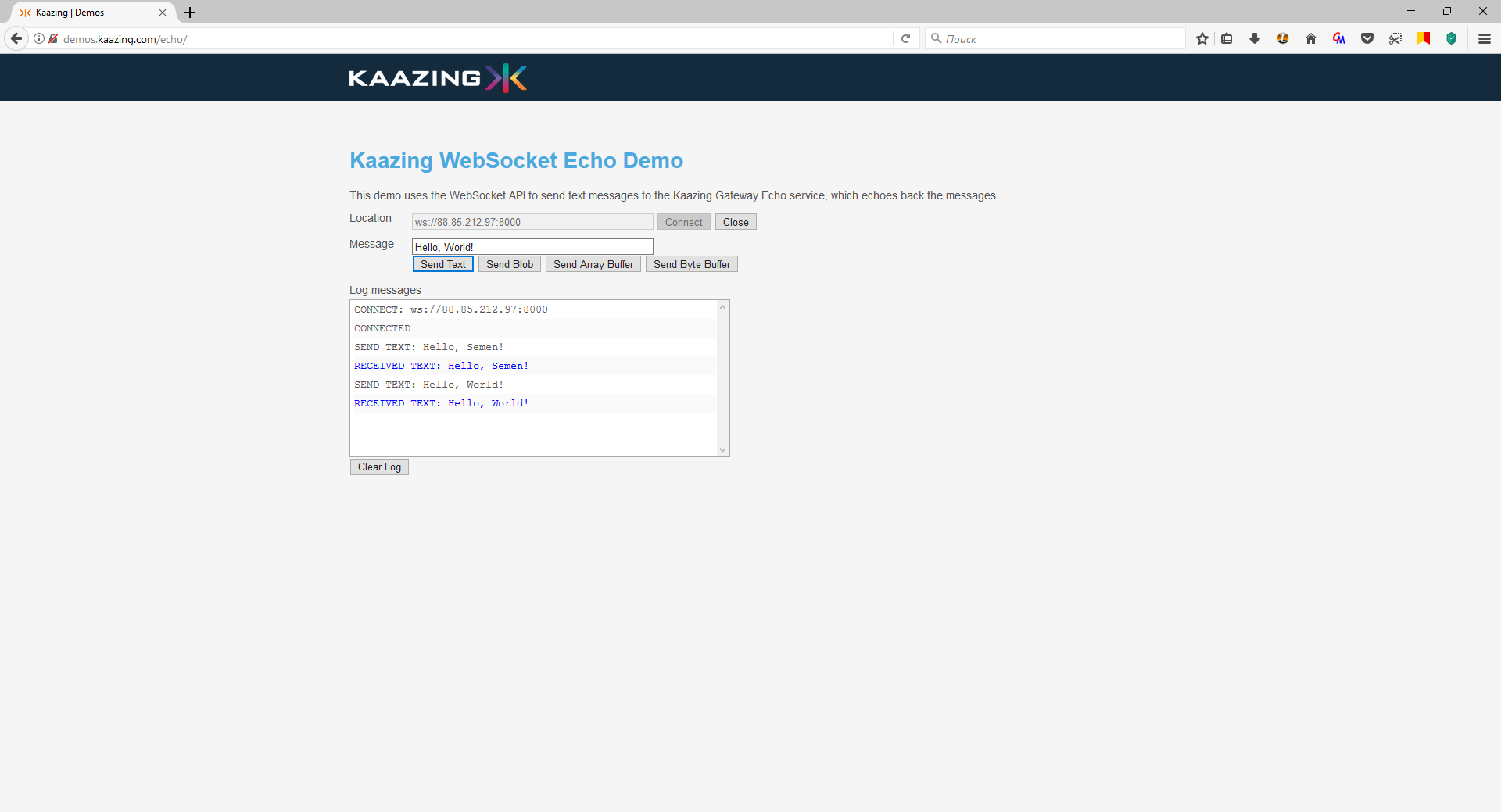
print("Ответ сервера:")

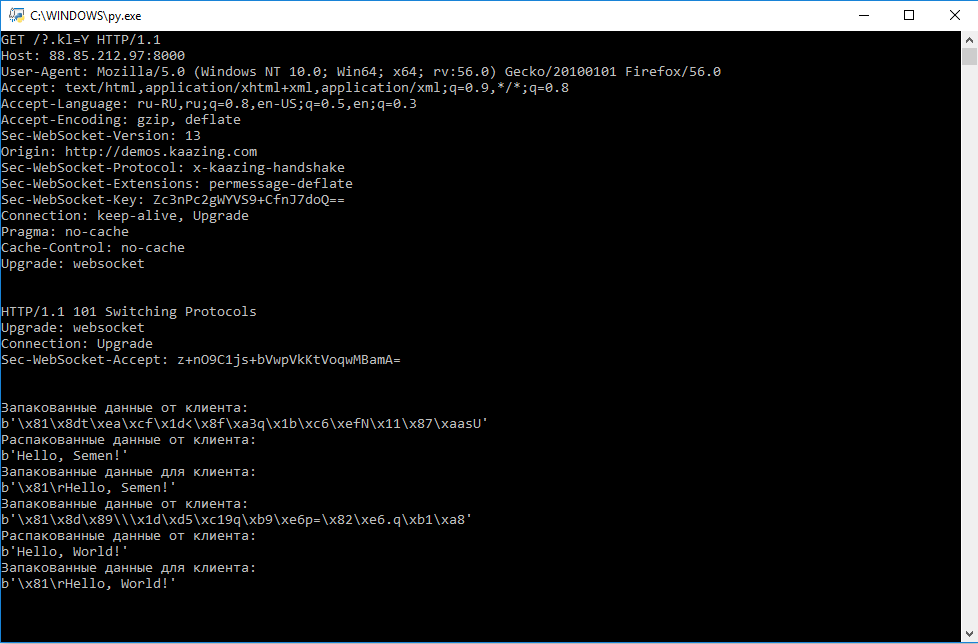
print(unpack\_frame(resp)['payload'])

#закрываем сокет

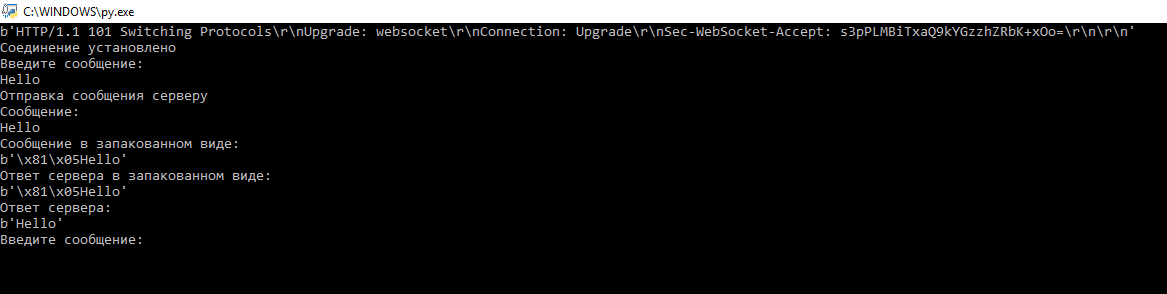
sock.close()

**Пример общения WebSocket сервера, код которого указан выше  
с клиентом** <http://demos.kaazing.com/echo/>

**Клиент:**

**Сервер:**

**Пример общения сервера и клиента на localhost:**

**Клиент:**

**Сервер:**

