Client-Server Network programming

Client-Server Architecture

- The **server**, that can be SW or HW appliance, provides a "service" necessary to one or more **clients**, or users of the service.

- Its sole reason for being is to wait for client requests, respond to these requests (and thus provide the service), and wait for other requests.

- Clients contact the server to get a certain service. They send the required data and wait for the response, which contains the data requested or a message that gives the reason of failure.

- While the server is running indefinitely, the client makes a single service request, get it, and closes the connection.

C/S network programming

- BEFORE the server can respond to client requests, you need to accomplish some preliminary operations. You must create the **endpoint** of a **communication channel** that allows the server to listen and wait for client requests.

- We must ensure that this communication channel is well known: think about a new web server ... without advertising remains unknown and unused.

- The client simply creates its endpoint and connects it to the server one: at this point the communication between client and server can start.

Socket: communication endpoints

- The sockets are the data structures that implement the concept of communication endpoints described above in the field of network communication between computers.

- Initially sockets were created to allow interprocess communication (IPC) between processes running on the same computer.

- There are two types of sockets, those based on files and those network oriented.

Common Sockets

- Unix sockets are the first family of sockets that we examine, and are called **AF_UNIX** or AF_LOCAL (Address Family: UNIX)

- Since both processes (client and server) are residing on the same host these sockets are based on **files**, so their structure is supported by the file system, which definitely makes sense because the file system is always shared between processes running on a given host.

- On a Linux system with this expression you can find all the file sockets.

```
find / -type s
```

- The second type of socket is based on the network and is called **AF_INET**, or AF_INET6 if use IPv6

- Python implements also AF_NETLINK for IPC between code in user and kernel space, and AP_TIPC (Transparent IPC) for communications between computer clusters without IP.

AF_INET Socket Addresses

- If a socket is the equivalent of a phone jack, a piece of infrastructure that enables communication, the **host name** and **port number** are like the area code and the phone number.

- The valid port numbers range from 0 to 65535, those up to 1024 are reserved for system use.
  - The ports in the range 0-1023 are called **well-known ports**. On unix-like operating systems, the opening of a port in this range to receive incoming connections requires administrator privileges. http://v.gd/denevi
  - The ports in the range 1024–49151 are called **registered ports**.
  - The ports in the range 49152–65535 are called **dynamic and/or private** ports.

Connection-Oriented Sockets

- You must **establish a connection** BEFORE that a communication can successfully start (virtual circuit or stream sockets)

- The connection-oriented communication offers the guarantee of data delivery. The primary network protocol that establishes a connection of this type is **TCP**.
  To create a TCP socket is necessary to create a **SOCK_STREAM** socket type

- AF_INET uses IP to find hosts in the network, and then the whole system SOCK_STREAM + AF_INET is indicated by the well-known acronym **TCP / IP**

- Naturally TCP can also be used with AF_UNIX, in which case it does not use IP.

TCP 3 way handshake

- To establish a connection, TCP uses 3 way handshake.
- Before a client can connect to a server, the server must **bind** itself to an IP port, open it and wait for incoming connection (passive open on 1 computer).
- Once the server passive open is done, the client may initiate an active session opening with the following process:
  1. The client sends a synchronization packet (**SYN**) to the server
  2. In response, the server send a "return receipt", with another syncronization packet (**SYN-ACK**)
  3. Finally the client sends the SYN return receipt (**ACK**)

TCP 3 way handshake

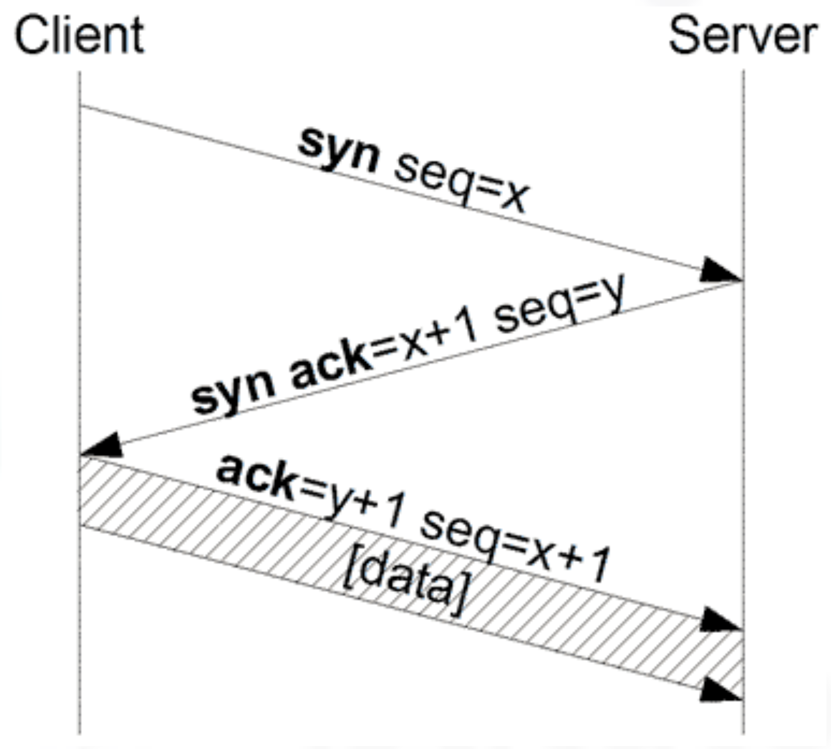
In details:

1. The client sends a SYN packet to the server to initialize the connection. The packet contains a random X sequence number

2. The server receives the packet, stores X and responds with a ACK that contains the next sequence number (X+1) expected. The server also starts the reverse connecction session, and then sends a SYN with the sequence number Y

3. The client responds with the next TCP package, with sequence number X+1, and the "receipt" of the Y package, which will contain the sequence number Y+1 expected.

TCP 3 way handshake



TCP connection

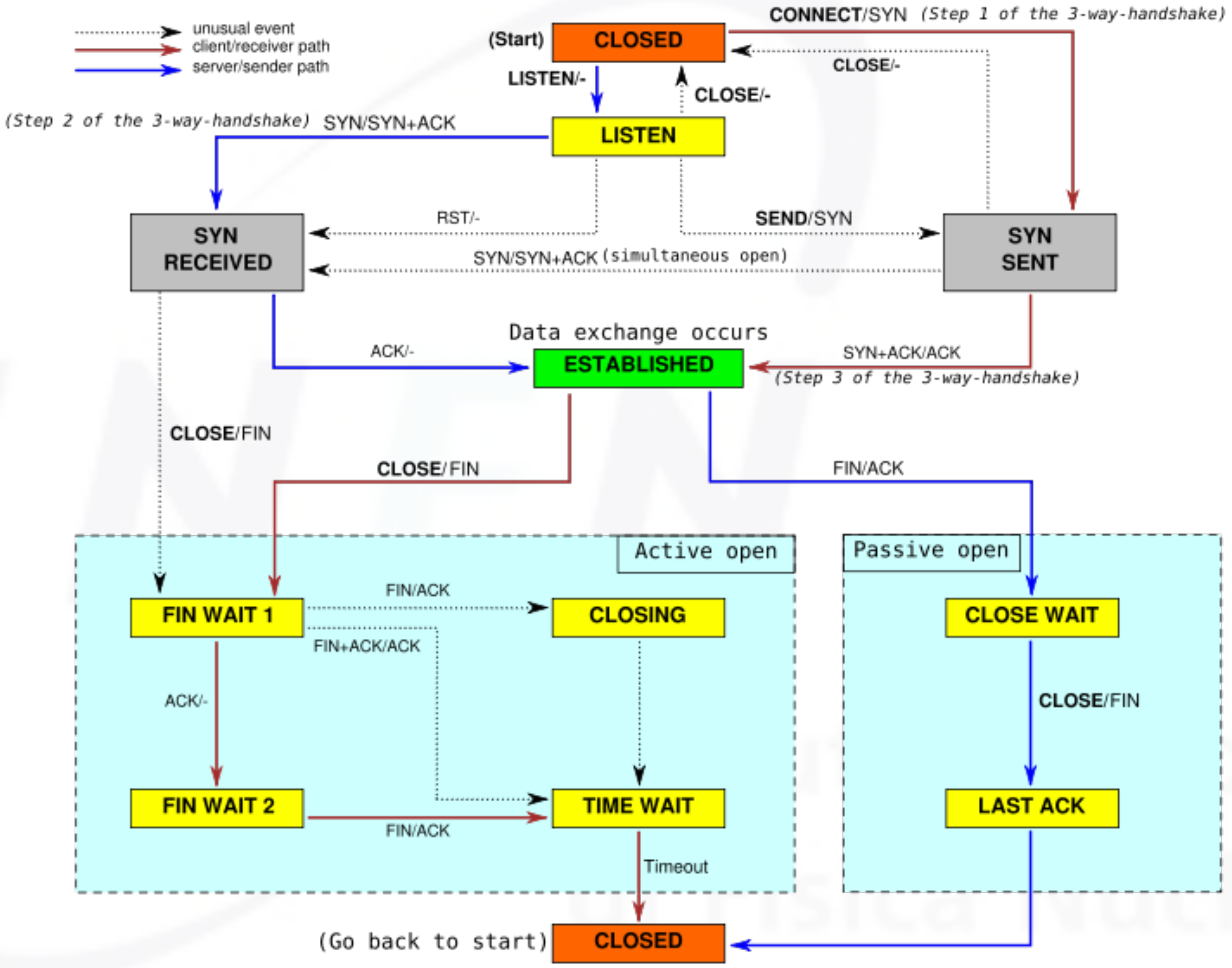
- Now the real connection can finally start (ESTABLISHED)
- As you may have guessed TCP uses a sequence number to identify the data, so that the data order is determined and the transmission reliable.
- TCP uses a "cumulative confirmed schedule": if the receiver acknowledges the receipt of the packet X, it means that he received any previous package. This ensures that packets arrive in order, that lost packets can be retransmitted and that there are no duplicates.

TCP connection

- To ensure the correctness of the transmitted data a **checksum** is entered in the packages (16 bits). It detects and compensates for simple errors introduced into the packets in communications between.
- TCP uses a **flow control** to prevent the sender to send data faster than the receiver is able to process (think of transferring data between a PC and a mobile phone)
- TCP uses several different mechanisms to control network congestion and prevent the collapse of performance. The ACK packets are used to understand what is the state of the network between the machines.

Connection closing

- TCP uses a "4-way handshake" to close connections; each side of the connection terminates independently.
- When one side wants to end the connection sends a **FYN** package, to which the other side must respond with an **ACK**; then the shutdown requires a couple of **FYN** and **ACK**
- A connection can be half opened, when only one side has finished
- You can also close with a 3 way handshake (FYN, FYN + ACK, ACK) if the client and server want to close the session simultaneously


unusual event
client/receiver path
server/sender path
(Start)
CLOSED
CONNECT/SYN (Step 1 of the 3-way-handshake)
CLOSE/-
LISTEN/-
CLOSE/-
(Step 2 of the 3-way-handshake) SYN/SYN+ACK
LISTEN
RST/-
SEND/SYN
SYN RECEIVED
SYN SENT
SYN/SYN+ACK (simultaneous open)
Data exchange occurs
ACK/-
ESTABLISHED
SYN+ACK/ACK
(Step 3 of the 3-way-handshake)
CLOSE/FIN
CLOSE/FIN
FIN/ACK
Active open
Passive open
FIN WAIT 1
FIN/ACK
CLOSING
FIN+ACK/ACK
CLOSE WAIT
ACK/-
CLOSE/FIN
FIN WAIT 2
FIN/ACK
TIME WAIT
LAST ACK
Timeout
(Go back to start)
CLOSED

Connectionless sockets

- The datagram sockets do not need to have a connection to start communication.
- In this case, however, there is no proper delivery guarantee: packets can be delivered in a different order from that in which they are sended, can not be delivered at all or can even be duplicated
- The primary protocol that establishes a connection of this type is **UDP**. To create a UDP socket is necessary to create a socket type **SOCK_DGRAM**
- AF_INET uses IP to find hosts in the network, and then the whole system SOCK_DGRAM + AF_INET is called **UDP / IP**

Differences between TCP and UDP

TCP ("Transmission Control Protocol") is connection-oriented, which means that it requires, before starting communication, a mutual confirmation of presence (handshake)

1. **Reliable:** TCP manages the confirmation of message reception, the retransmission of the lost ones and the timeout. Many attempts are made to properly deliver each packet: if it is lost on the way, the server requests a resend. In TCP connections any data is correctly sended, or there are multiple time-out and the connection is dropped.

2. **Ordered:** If two packets are sent on a connection, one after another, the first one arrives first to the destination host. When the packets arrive in the wrong order, TCP keeps waiting until the data is not able to reconstruct the chain in the correct temporal order.

3. **Heavy**: TCP requires three packets only to initialize the connection ... It manages the connections, the reliability and congestion control.

4. **Streaming:** The data are read as a stream, the packets may be divided or grouped according to the need.

Differences between TCP and UDP

UDP is a simpler protocol, without support for connections, ie there is no mechanism to ensure that between the two hosts there is an open communication channel. One host sends information in one direction, without checking whether the recipient is still there, or if it is ready to receive messages.

1. **Unreliable**: When a message is sent it is not possible to know if it will reach its destination. There is no confirmation of receipt, neither retransmission nor timeout.

2. **Unorderes**: If two messages are sent to the same recipient you can not predict their order of arrival.

3. **Light**: There are not control structures

4. **Datagrams**: Packets are sent individually and are guaranteed to be whole when at destination. You can not divide or unite the packages into smaller or larger structures

Python Socket

`socket(`*`socket_family`*`, `*`socket_type`*`, `*`protocol`*`=0)`

where	*socket_family	= AF_UNIX | AF_INET
	*socket_type	= SOCK_STREAM | SOCK_DGRAM

- To create a TCP / IP socket:

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

- To create a UDP / IP socket:

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

- To avoid the use of prefix *socket* remember that you can use:

  ```
  from socket import *

  tcpSock=socket(AF_INET, SOCK_STREAM)
  ```

TCP Server

```
ss = socket()                    # create server socket
ss.bind()                        # bind socket to address
ss.listen()                      # listen for connections
inf_loop:                        # server infinite loop
    cs = ss.accept()             # accept client connection
    comm_loop:                   # communication loop
        cs.recv()/cs.send()      # dialog
    cs.close()                   # close client socket
ss.close()                       # close server socket # (opt)
```

All sockets are created with `socket.socket()` function

The servers must "sit on a door" and wait for requests, so they must bind to a local address

Since TCP is a connection-oriented communication system, you need to prepare infrastructure before starting communication.

A simple single threaded server will always be in a state of `accept()`, waiting for the client communication. Accept is typically blocking, the execution is suspended until the connection arrives. There is also a non-blocking version.

Once the connection is accepted `accept()` returns a client socket on which communication takes place. If you use the SocketServer module you can delegate this part to a new thread or process, allowing the main loop to resume.

When the client socket is closed the server continues to wait. The last line is optional, it should never be used, unless you have a condition for which the server must be stopped and must get out smartly http://bit.do/TCPServer-py

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

from socket import *
from time import ctime

HOST = ''
PORT = 21567
BUFSIZ = 1024
ADDR = (HOST, PORT)

tcpSerSock = socket(AF_INET, SOCK_STREAM)
tcpSerSock.bind(ADDR)
tcpSerSock.listen(5)

while True:
    print 'waiting for connection...'
    tcpCliSock, addr = tcpSerSock.accept()
    print '...connected from:', addr
    while True:
        data = tcpCliSock.recv(BUFSIZ)
        if not data:
            break
        tcpCliSock.send('[%s] %s' % (ctime(), data))
    tcpCliSock.close()
tcpSerSock.close()
```

Any available address

Random port

Maximum number of connections accepted

Infinite loop waiting for client connection. When it arrives the loop of communication starts.
The server replies with the same message received + timestamp.

This line should never be executed

Graceful exit

To catch the SIGINT message and exit with no error message the servers must define a **handler** of the message, and within it include the correct socket shutdown. The code can be inserted before the while loop of the listening loop.

```
http://bit.do/TCPServer2-py
```

```
import signal
import sys
def signal_handler(signal,frame):
     print 'You pressed Ctrl+C…exiting'
     tcpSerSock.close()
     sys.exit(0)

signal.signal(signal.SIGINT, signal_handler)
```

Client TCP

```
cs = socket()                 # create client socket
cs.connect()                  # attempt server connection
comm_loop:                    # communication loop
   cs.send()/cs.recv()        # dialog
cs.close()                    # close client socket
```

Create a client is much easier than a server.

After creating a socket the client can use it to request a connection to the server, and enter the loop of communication.

Finally at the end of the communication the socket is closed

```
#!/usr/bin/env python
from socket import *
import signal,sys

HOST = 'localhost'
PORT = 21567
BUFSIZ = 1024
ADDR = (HOST, PORT)
def signal_handler(signal, frame):
    print 'You pressed Ctrl+C...exiting'
    sys.exit(0)

signal.signal(signal.SIGINT,signal_handler)

while True:
    tcpCliSock = socket(AF_INET, SOCK_STREAM)
    tcpCliSock.connect(ADDR)
    data = raw_input('> ')
    if not data:
        break
    tcpCliSock.send('%s\r\n' % data)
    data = tcpCliSock.recv(BUFSIZ)
    if not data:
        break
    print data.strip()
tcpCliSock.close()
```

http://bit.do/TCPClient-py

The address is now referred to the SERVER!

Infinite loop, that stops if the user does not enter anything or if the server is not responding and then `recv()` fails.

You can try the client and server, and make sure they work

UDP Server and client

```
ss = socket()                            # create server socket
ss.bind()                                # bind server socket
inf_loop:                                # server infinite loop
    cs = ss.recvfrom()/ss.sendto()       # dialog
ss.close()                               # close server socket (opt)
```

The server does not need to prepare the connection control structures: once it has the socket on the couple host / port, the infinite loop of dialog can start

```
cs = socket()                            # create client socket
comm_loop:                               # communication loop
    cs.sendto()/cs.recvfrom()            # dialog
cs.close()                               # close client socket
```

```
#!/usr/bin/env python
from socket import *
from time import ctime
import signal, sys

HOST = ''
PORT = 21567
BUFSIZ = 1024
ADDR = (HOST, PORT)

udpSerSock = socket(AF_INET, SOCK_DGRAM)
udpSerSock.bind(ADDR)

def signal_handler(signal, frame):
    print 'You pressed Ctrl+C...exiting'
    udpSerSock.close()
    sys.exit(0)

signal.signal(signal.SIGINT,signal_handler)

while True:
    print 'waiting for message...'
    data, addr = udpSerSock.recvfrom(BUFSIZ)
    udpSerSock.sendto('[%s] %s' % (ctime(), data), addr)
    print '...received from and returned to:', addr

udpSerSock.close()
```

http://bit.do/UDPServer-py

The loop is more simple: expect passively the message (datagram), when it arrives process it by adding the time stamp, then send it back, and start over waiting for a new message. The handshake procedure and the creation of a dedicated socket is missing http://bit.do/UDPClient-py

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

from socket import *

HOST = 'localhost'
PORT = 21567
BUFSIZ = 1024
ADDR = (HOST, PORT)

udpCliSock = socket(AF_INET, SOCK_DGRAM)

while True:
    data = raw_input('> ')
    if not data:
        break
    udpCliSock.sendto(data, ADDR)
    data, ADDR = udpCliSock.recvfrom(BUFSIZ)
    if not data:
        break
    print data

udpCliSock.close()
```

You can try the client and server, and make sure they work

SocketServer module

- It is a high-level module, which greatly simplifies the details of the creation of a server and a network client.

- Much of the "dirty work" that we have seen so far is done behind the scenes.

- The module interface uses classes.

- The module programming is "*event driven*", the event to which the program reacts is the arrival of a message http://bit.do/SSTCPServer-py

```
#!/usr/bin/env python
from SocketServer import (TCPServer as TCP, StreamRequestHandler as SRH)
from time import ctime
import signal,sys

HOST = ''
PORT = 21567
ADDR = (HOST, PORT)

def signal_handler(signal, frame):
    print 'You pressed Ctrl+C...exiting'
    sys.exit(0)
signal.signal(signal.SIGINT,signal_handler)

class MyRequestHandler(SRH):
    def handle(self):
        print '...connected from:', self.client_address
        self.wfile.write('[%s] %s' % (ctime(), self.rfile.readline()))

tcpServ = TCP(ADDR, MyRequestHandler)
print 'waiting for connection...'
tcpServ.serve_forever()
```

The incoming messages handler is derived from SRH, and rewrites the `handle()` function (the default contains `pass`), that is called when the server receives a message

Sockets are managed as "file" objects

http://bit.do/SSTCPClient-py

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

from socket import *
HOST = 'localhost'
PORT = 21567
BUFSIZ = 1024
ADDR = (HOST, PORT)

while True:
     tcpCliSock = socket(AF_INET, SOCK_STREAM)
     tcpCliSock.connect(ADDR)
     data = raw_input('> ')
     if not data:
          break
     tcpCliSock.send('%s\r\n' % data)
     data = tcpCliSock.recv(BUFSIZ)
     if not data:
          break
     print data.strip()
     tcpCliSock.close()
```

The SS handler's default behavior is to accept the connection, read the request and then close the connection. In the loop now you must open and close single connections. You can change this behavior rewriting the default methods of the handler function.

You need to introduce the line end.