Network Programming

Read beyond the textbook

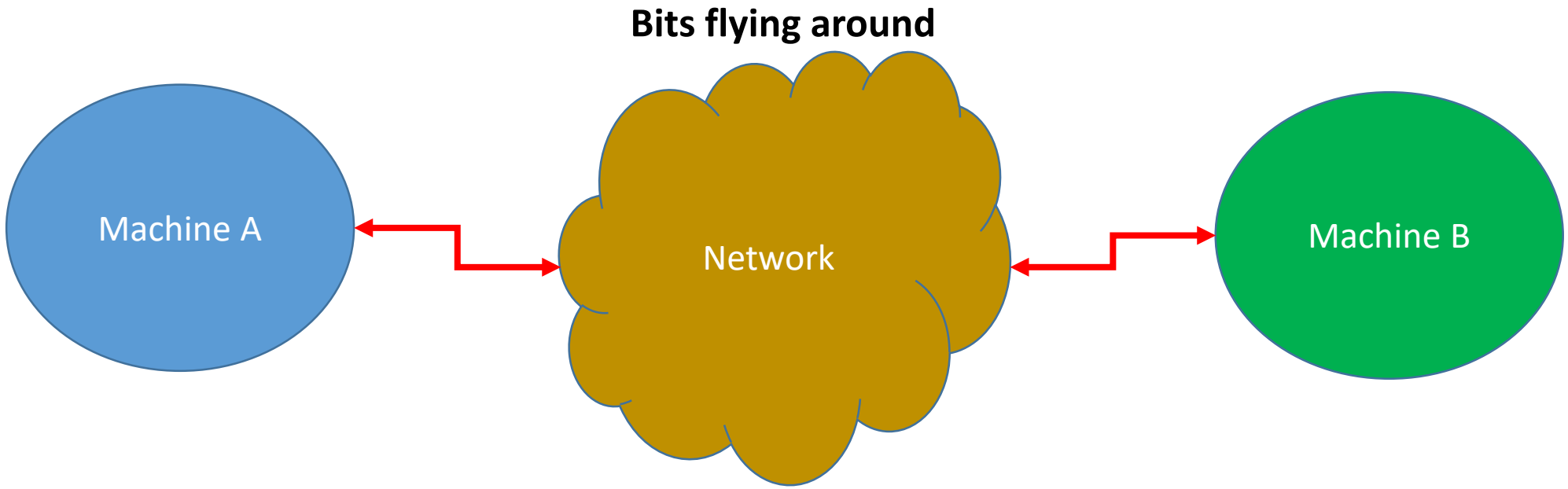
Python Network Programming

- Network Programming is a major use of Python
- Python standard library has wide support for network protocols, data encoding/decoding, and other things you need to make it work
    - Python 3 makes you do everything in bytes
- It is much easier to do network programming in Python than any other language like C/C++.

Networking – extending beyond your OS

- Go from network basics using sockets
- To advanced networking using server classes with concurrent execution
- On the way, we will implement FTP

Here's the problem – communication between machines or computers



Term used: message passing

Two kinds of ports: Physical or Virtual

- Physical ports:
  - Physical Cable connections:
    - Serial, Ethernet, usb, etc.
- Virtual ports – essential for IP networking
  - These ports allow software applications to share hardware resources without interfering with each other.
  - Logical constructs used in software
  - This is the kind we use for networking

Network Addressing

Machines are addressable by two parameters:

- **Machine name and IP (Internet Protocol) address**
- **Programs/services and port numbers**

A port in computer networking is a logical access channel for communication between two devices.

A protocol is a set of rules.

In computer networking, a protocol defines a standard way for computers to exchange information.

Most common protocols used in computer networks and the internet are TCP (Transmission Control Protocol), UDP (User Datagram Protocol), and IP (Internet Protocol).

Port numbers are generally divided into three ranges:

1. **The Well Known ports: 0 to 1023**
2. **The Registered ports: 1024 to 49151**
3. **The Dynamic and/or Private ports: 49152 to 65535**

https://www.speedguide.net/ports.php

Standard Ports

Port	Service
21	FTP
22	SSH
23	Telnet
25	SMTP (Mail)
80	HTTP (Web)
110	POP3 (Mail)
443	HTTPS (Web)

Other port numbers may be randomly assigned to programs by the Operating System

Windows:

cmd window: resmon.exe

MacOS:

Network Utility

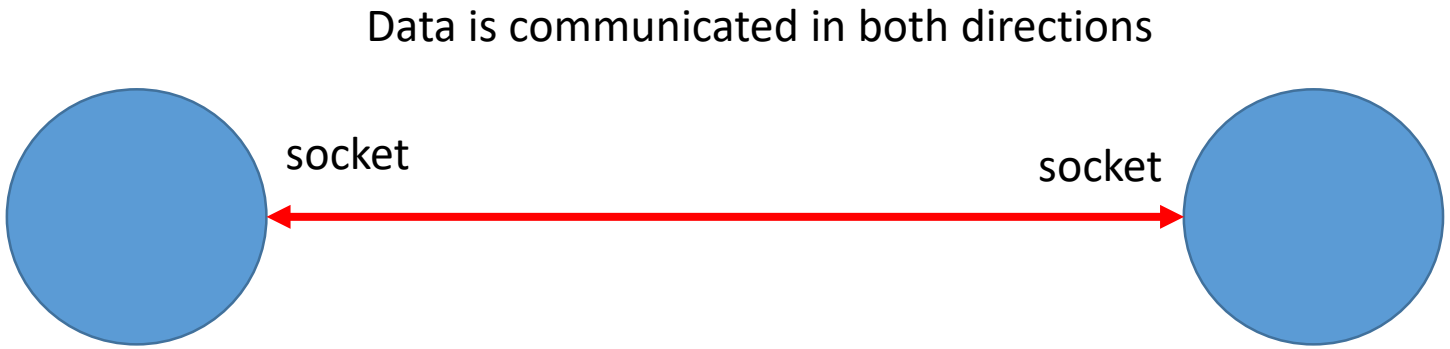
http://osxdaily.com/2013/10/31/use-network-utility-mac-os-x/

Connections

- A network connection is always represented by a host and port number
- In python, it is written as a tuple (host, port)
  - (‘www.python.org’, 80)
  - (‘122.172.13.4’,4430

Sockets

- A programming abstraction for network connection
- Socket is considered an endpoint of a network connection



Data Transport – two forms

- Streams (TCP):
  - Computers establish a connection with each other and read/write data in a continuous stream of bytes---like a file. This is the most common.
- Datagrams (UDP):
  - Computers send discrete packets (or messages) to each other. Each packet contains a collection of bytes, but each packet is separate and self-contained.

Let’s do some coding

server1.py and client1.py

server2.py and client2.py

Client1.py

```
s = socket.socket()
```

```
ahost = '127.0.0.1'
# connect to the server on local computer
s.connect((ahost, port))
```

```
aa = s.recv(1024)
aa.decode('utf-8')
```

```
c.close()
```

Server1.py

```
port = 12345
s = socket.socket()
s.bind(('', port))
s.listen(5)
```

Normal way
To setup a server

s is known as the 'well-known socket' because its IP address and port is known by everyone

```
while True:
    c, addr = s.accept()
    str = 'You are connected'
    c.send(bytes(str,'utf-8') )
    c.close()
```

Takes one Client At a time

c is the client socket

MS VSC only allows one process to run using “run python file in terminal”.

To run the client program, create a new cmd terminal in VSC, and manually Execute the script client1.py

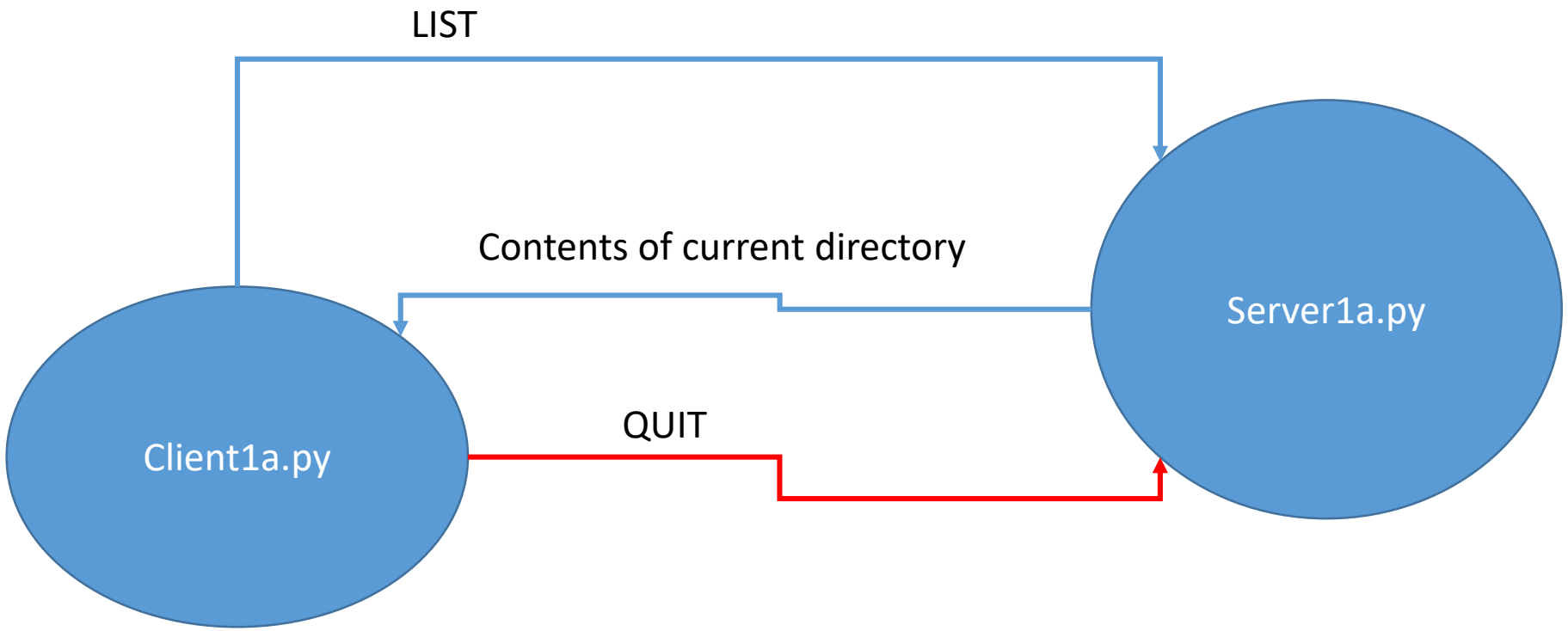
OR to create a normal terminal window.

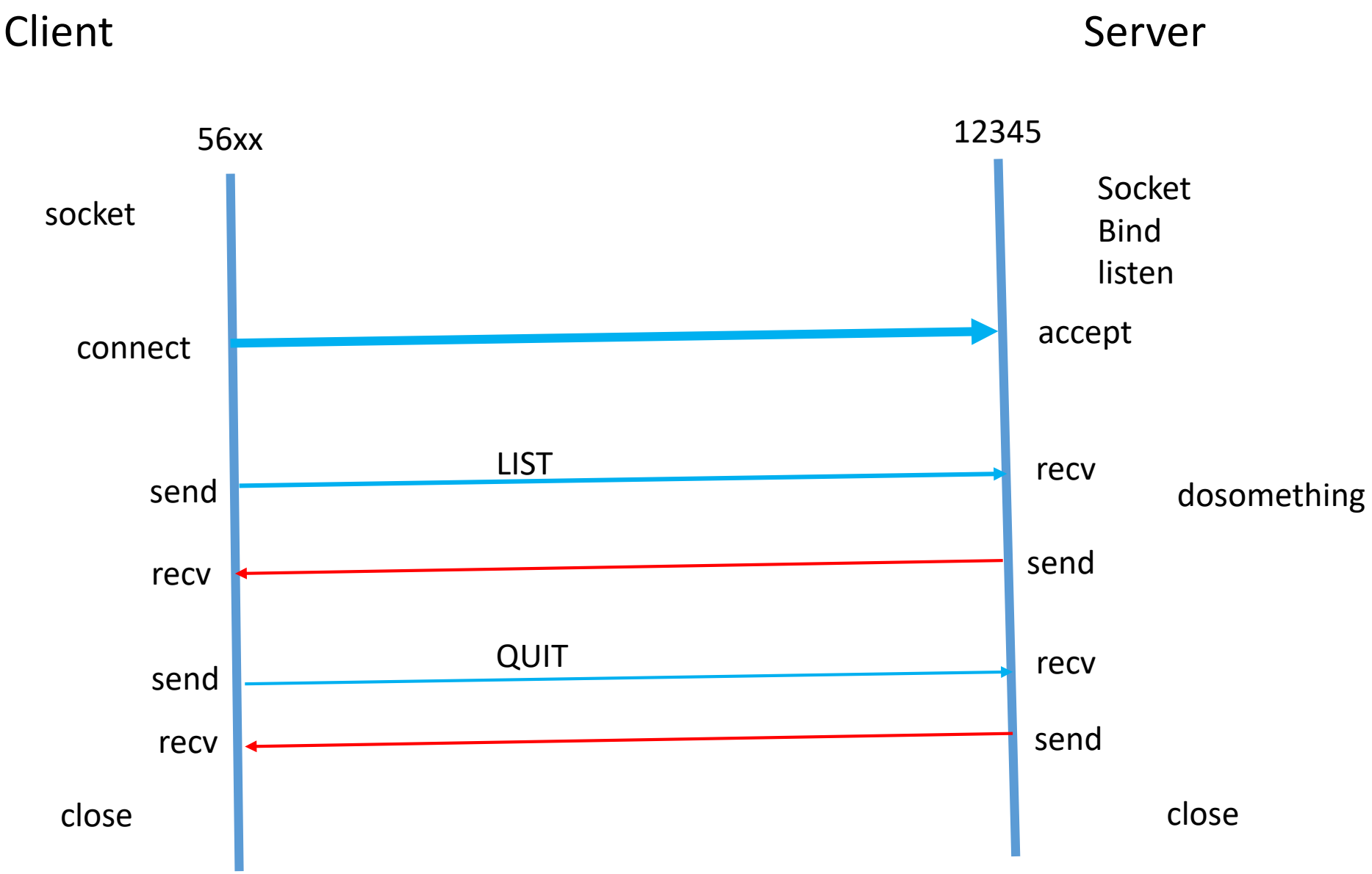
Class Exercise

Modify client1.py and server1.py => client1a.py and server1a.py

Client sends a LIST cmd
And server to respond with a listing of the content of the current directory once it sees the LIST cmd

Client sends a “QUIT” cmd, and server to close connection.
Client to close connection.


LIST
Contents of current directory
QUIT
Client1a.py
Server1a.py


Client
Server
56xx
12345
socket
Socket
Bind
listen
connect
accept
LIST
send
recv
dosomething
recv
send
QUIT
send
recv
recv
send
close
close

Test client1a.py and server1a.py

- Each person’s client1a.py to connect to another person’s server1a.py
- Get the IP address of server1a.py

UTF-8 is a variable width character encoding capable of encoding all 1,112,064[1] valid code points in Unicode using one to four 8-bit bytes

UTF-8 is the "mandatory" Unicode character encoding[7] for the World Wide Web;

The first 128 characters (US-ASCII) need one byte.

Server2.py

Initial setup
… create socket
… bind socket
… listen socket

While True:
accept connection
create a ClientThread
start the clientThread
add to the list of threads

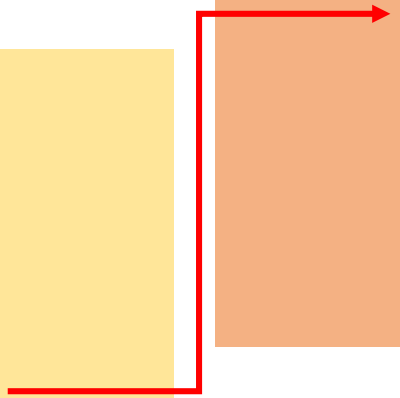
Class ClientThread(Thread):
def __init__

def run():
while True:
recv
parse cmd from string
result = goCmd(cmd)
send result back

Client2.py

Create socket
Connect

sendCmd('LIST')
….



More Sockets

- Socket programming is usually not elegant and ad hoc
- A number of options in the use of socket
- Number of failure modes which you have to take care
- Will cover some critical issues here

Partial reads/writes

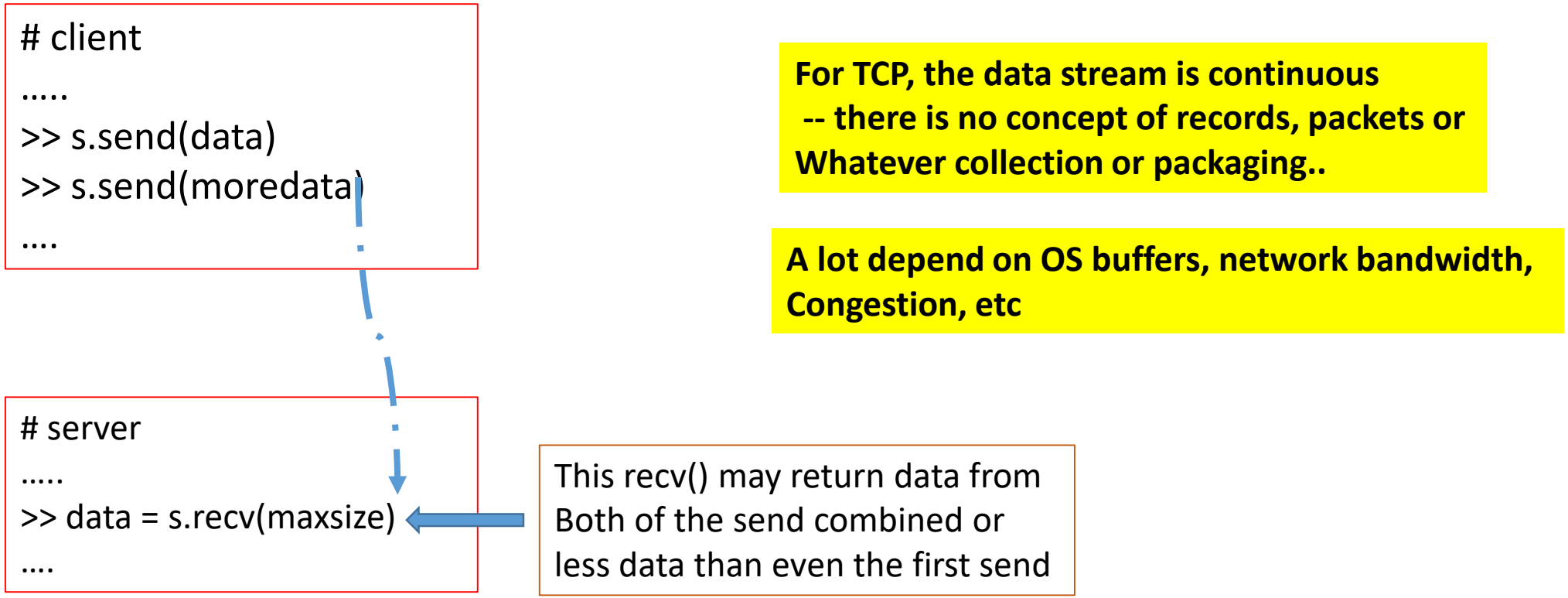
- WARNING: reading/writing to a socket may involve partial data transfer
- send() returns actual bytes sent
- recv() length is only a maximum limit
- Eg.
  - >>> len(data)
  - 200000
  - >>> s.send(data)
  - 45678

```
>>> len(data)
200000
>>> s.send(data)
45678
```

Sent partial data

```
>>> data = s.recv(100000)
>>> len(data)
65678
```

Received less than max



For TCP, the data stream is continuous -- there is no concept of records, packets or Whatever collection or packaging..

A lot depend on OS buffers, network bandwidth, Congestion, etc

s.sendall(data) – Sending all data

- Wait until all data is sent, use sendall()
  - s.sendall(data)
- Blocks until all data is transferred
- For most applications this is what you would normally do
- HOWEVER, if you are doing screen updates, multitasking and other multiprocessing tasks, this may not be the best choice

send() is asynchronous while sendall() blocks until everything is sent.
send() is returned immediately and continues to send in the background

End of Data

- How to tell if there is no more data?
- recv() will return empty string
  - >> data = s.recv(1000)
  - >> len(data)
  - 0
  - >>>
- This means that the other end of the connection has been closed (no more sends)

On the receiver side – do some Data Reassembly

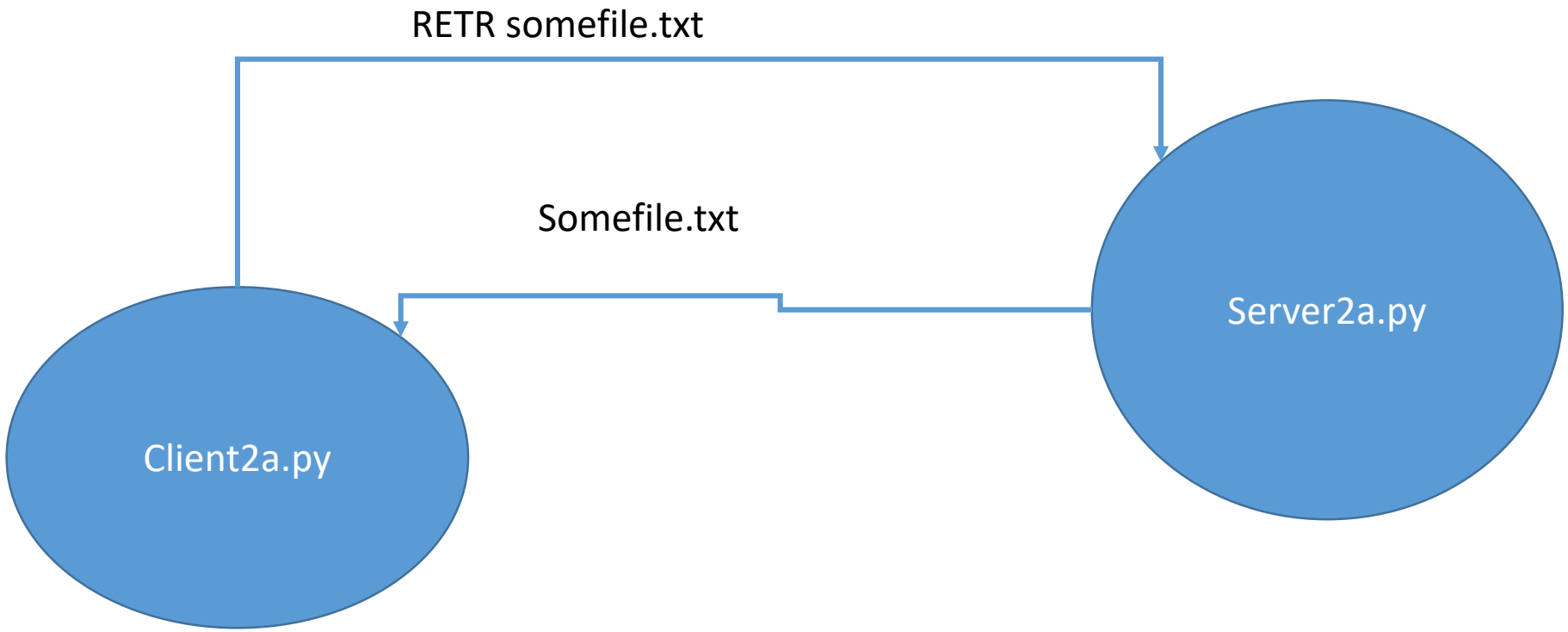
- Receivers often need to reassemble messages from a series of small chunks

```
chunklist = []          # list of chunks
while not done:
      chunk = s.recv(maxsize)   # read some chunk
      if not chunk:
           break
       chunklist.append(chunk)

# finally reassemble the message
message = ‘’.join(chunklist)
```

Code template for doing Data reassembly

GOAL: Copy a big text file from server to client.

Exercise: Implement RETR somefile.txt



Make copies of client2.py and server2.py into client2a.py and server2a.py
Modify them to retrieve a file somefile.txt and save it as afile.txt

FTP

Build something useful

```
# read the file line by line and send it line by line
def retrieveFile(self, filename):
        with open(filename, 'r') as fd:
                for aline in fd:
                        self.sock.send(bytes(aline,'utf-8'))
```

Server side

```
def receiveFile():
        fd = open('retrfile.txt','w')
        while True:
                try:
                        s.settimeout(2)
                        chunk = s.recv(maxsize) # read some chunk
                        print('recv file line: ', chunk)
                        if chunk == b'' or not chunk:
                                break
                        else:
                                fd.write(chunk.decode('utf-8'))
                except Exception as e:
                        print("Exception: ", e)
                        s.settimeout(0)
                        s.setblocking(True)
                        break
        fd.close()
```

Timeouts

- Most socket operations block indefinitely
- Can set an optional timeout
  - >> s = socket() # default is socket(AF_INET, SOCK_STREAM)
  - …
  - >> s.settimeout(5.0) # timeout of 5 seconds
  - …
- This will result in a timeout exception
- To disable:
  - >> s.settimeout(None)

More options and features for sockets

- Find them out yourself
- Socket programming is the first step towards writing programs that run on other machines
- First step towards distributed something…
  - Distributed file system
  - Distributed systems
  - Distributed computing
  - Distributed OS
  - ….

One of your classes

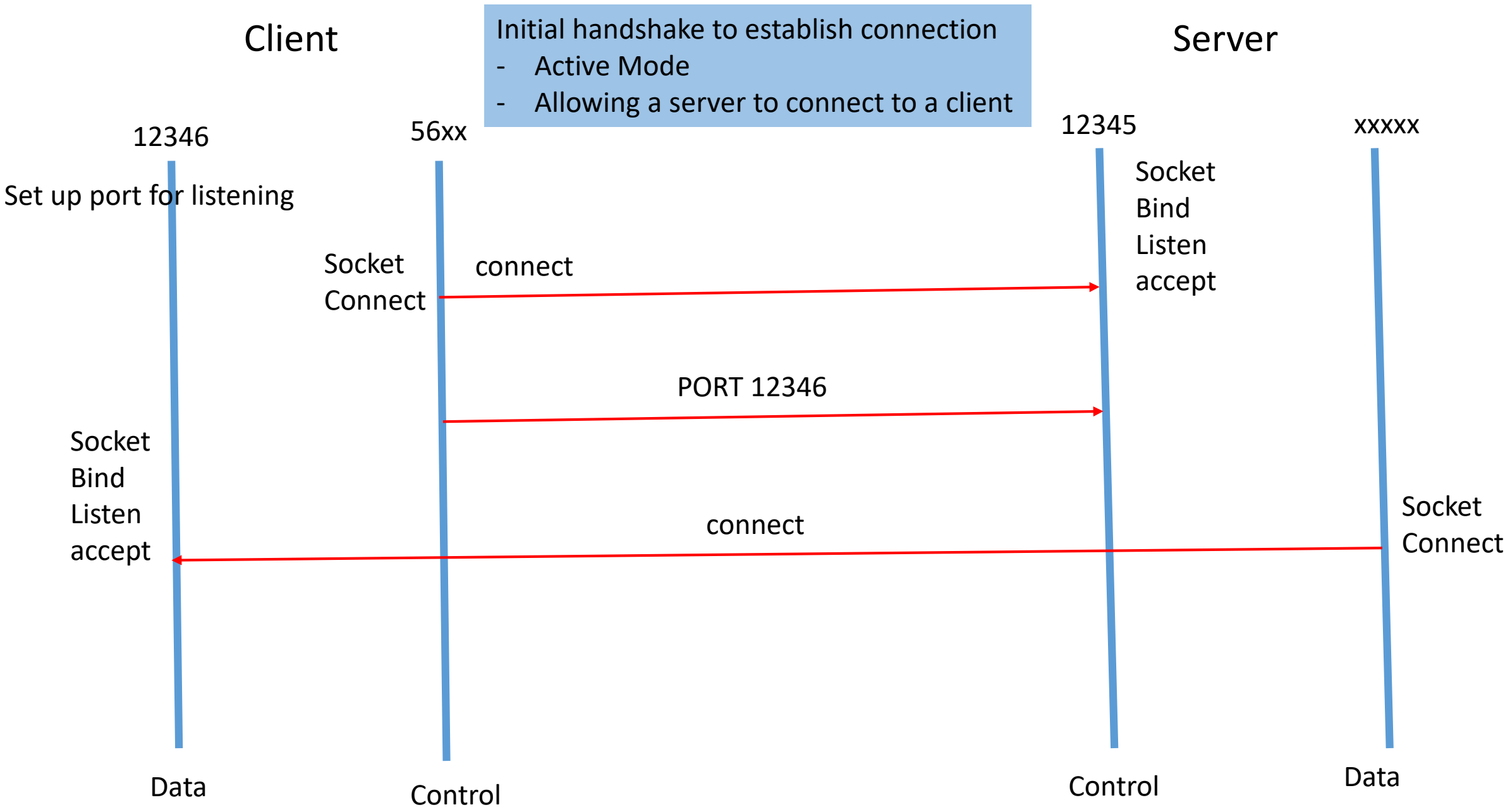
Class exercise:

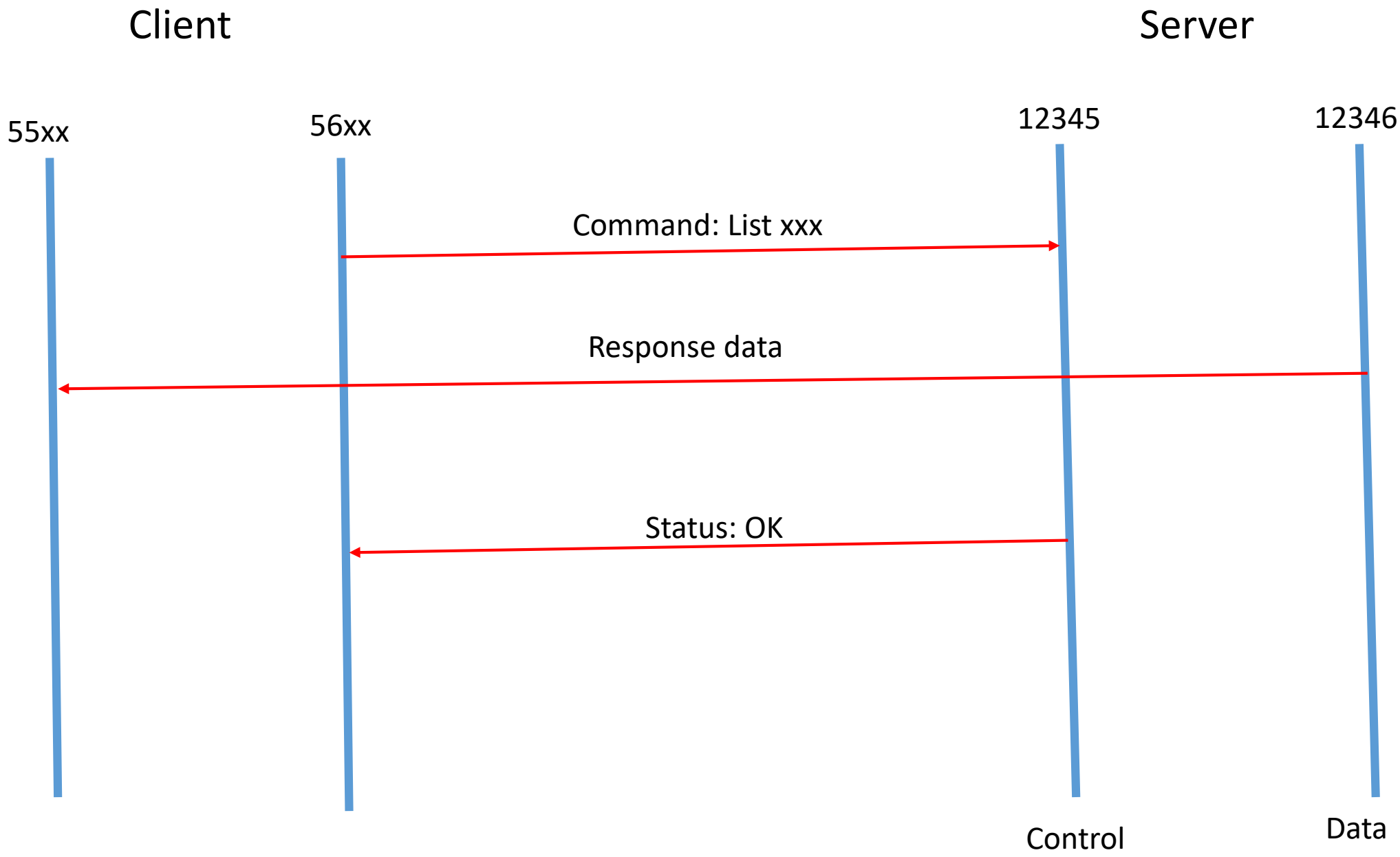
- Server establishes the control port
  - One socket – called the control port – listens for connections and commands
    - port: 12345
- Client establishes the data port
  - Set up a listening port – called the data port: 12346
  - Connect to server at the control port: 12345
  - Once connected, send this command ‘PORT 12346’ to server
  - Server will attempt to establish connection to this data port
- Client ready to send commands
  - Send a command to control port – something like “LIST’
    - Server sends a response data to data port
    - Server sends status: ‘OK’
  - Close all connections

RFC 959

USE raw sockets and not any other library

ftpclient.py and ftpserver.py


Client
Initial handshake to establish connection
- Active Mode
- Allowing a server to connect to a client
Server
12346
56xx
12345
xxxxx
Set up port for listening
Socket
Bind
Listen
accept
Socket
Connect
connect
PORT 12346
Socket
Bind
Listen
accept
connect
Socket
Connect
Data
Control
Control
Data


Client
Server
55xx
56xx
12345
12346
Command: List xxx
Response data
Status: OK
Control
Data

ftpclient.py

```
Establish socket
cs = connect()
msg = cs.recv()
cs.send('PORT 12346')
ds = socket()
 ds.bind(IP, 12346)
ds.listen(1)

sendCmd(LIST)
```

ftpserver.py

```
Class NewClientThread

def run():
    recv PORT message]
    ds = connect to client's port
    #ds is the dataSocket
    send status to cs
    while True:
        cmd = cs.recv()
        result =  doCmd()
        ds.send(result)
        cs.send('OK')
```

Establish socket and listen

```
While loop
    cs = accept connection
     # cs is controlSocket
    cs.send('connected')
    create a NewClientThread
    start new Thread
```

Two solutions to the send and receive file

- Using a timeout on the receive side
  - Socket.settimeout(n)
  - Socket.recv(m)

Use the timeout feature for this implementation

- Now that you have a control channel
  - Send the size of the file via control channel
  - Then send the data via data channel

```
filesize = str(os.path.getsize(path))
```

    - controlsock.send(bytes([filesize],'utf-8'))
    - Keep track of bytes received.
    - If difference between filesize and totalbytes received is zero, DO not do any more receive

FTP – File Transfer Protocol

Took nearly 14 years
For ftp to stabilize,
Yet we are still finding issues..

- The original specification for the File Transfer Protocol was written by Abhay Bhushan and published as RFC 114 on 16 April 1971.
- FTP may run in *active* or *passive* mode, which determines how the data connection is established. In both cases, the client creates a TCP control connection from a random, usually an unprivileged, port N to the FTP server command port 21.
  - A Request for Comments (**RFC**) is a formal document from the Internet Engineering Task Force (IETF) that is the result of committee drafting and subsequent review by interested parties. Some **RFCs** are informational in nature.
- I have included a document on FTP. – RFC 695. 1985

Client shell	Command to send	Behavior
cd	CWD xxx	Change working remote directory
pwd	PWD	display current remote directory
lcd	none	Change local directory
ls	LIST	Return contents of remote directory
get	RETR xxx	Return copy of file
system	SYST	Return system type
put	STOR xxx	Send a file to remote directory
bye	QUIT	Close connection
connect	none	Connect to remote host in active mode
exit	none	Exit the program

https://en.wikipedia.org/wiki/List_of_FTP_commands

Error codes returned

FTP server should be sending back status codes:
200
500
etc

```
200 Command okay.
500 Syntax error, command unrecognized.
     This may include errors such as command line too long.
501 Syntax error in parameters or arguments.
202 Command not implemented, superfluous at this site.
502 Command not implemented.
503 Bad sequence of commands.
504 Command not implemented for that parameter.
```

https://tools.ietf.org/html/rfc959

https://www.smartfile.com/blog/big-list-ftp-server-response-codes/

```
while in_command_loop:
    raw_input = input("Prompt>")
    if input == 'command_1':
        command_1()
    elif input == 'command_2':
        command_2()
    # more commands here...
    else:
        print "Error"
```

```
from cmd import Cmd


class MyPrompt(Cmd):


    def do_hello(self, args):
        """Says hello. If you provide a name, it will greet you with it."""
        if len(args) == 0:
            name = 'stranger'
        else:
            name = args
        print "Hello, %s" % name


    def do_quit(self, args):
        """Quits the program."""
        print "Quitting."
        raise SystemExit
```

For building a shell:

https://wiki.python.org/moin/CmdModule

https://docs.python.org/3/library/cmd.html

SINCE you are going to run your FTP-client against another FTP-server,
We should stick to one protocol for send and receive file.

For server: just send or sendall
For receiver: use a timeout of 2 or three seconds.
on timeout exception – that will be the end of file
set timeout back to 0
setblocking to True

Let's talk about Software Design

The top-down approach helps to make your code cleaner

```
def do_connect()
    parse the arguments
    conn = MyConnection(ip, port)
    if conn.connect():
        print('good')
    else:
        print('not good')
```

```
class MyConnection():

    def __init__(self, ip, port):
        … variables

    def connect(self):
        socket
        controlport = connect

        set up data port
            datasocket = socket.socket()
            bind
            listen on datasocket
        send port cmd via controlport
        accept
        return good or bad
```

Let's try a get file

```
def do_get():
    parse the arguments
    if conn.get_file(filename):
        print('good – file received')
    else:
        print('bad')
```

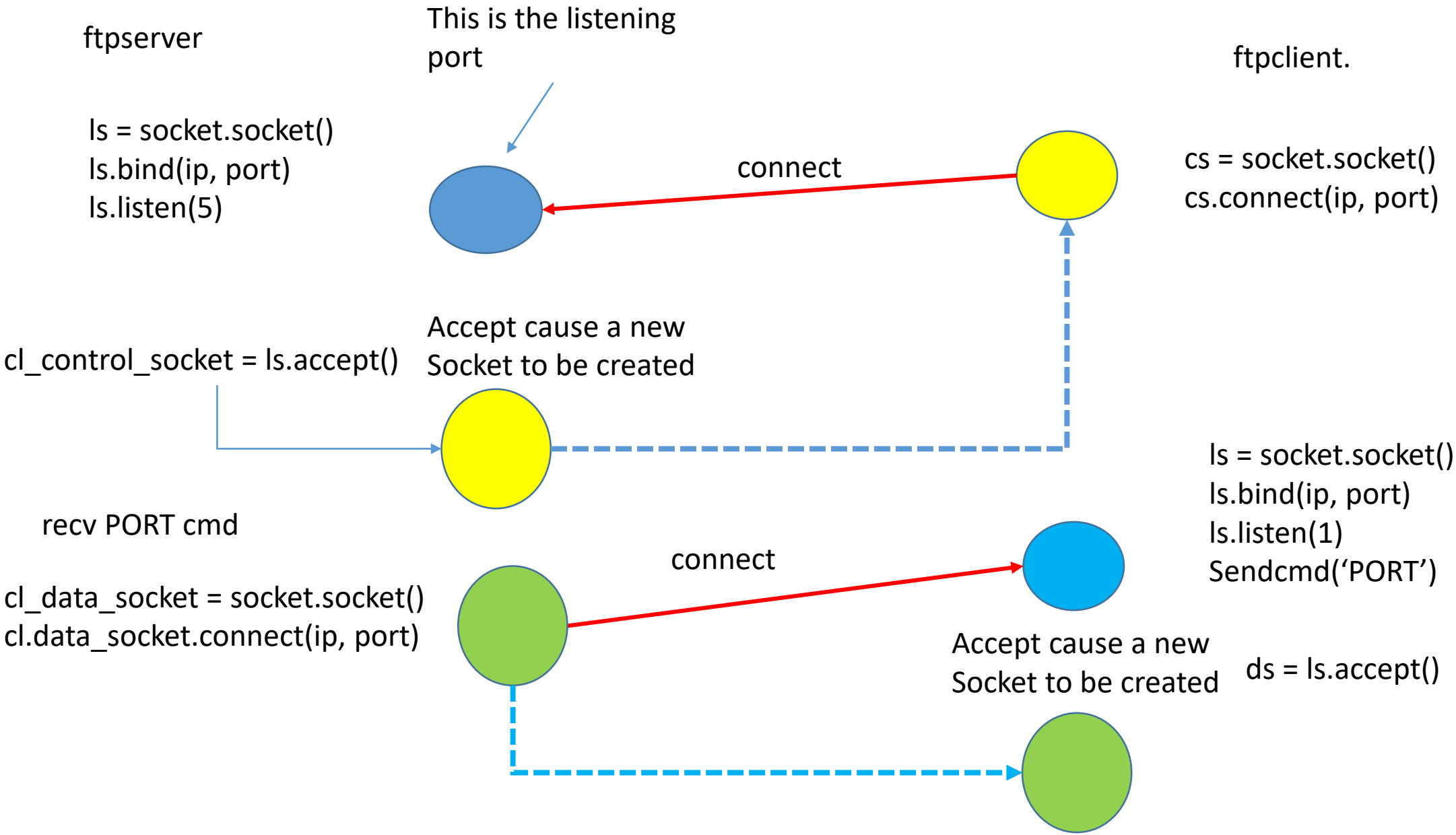
```
Class MyConnection():

    def get_file(self, filename):
        send RETR cmd via control socket
        do receive with timeout
        return good or bad
```

Main idea: build up the top components first, and then build the lower level components

Client shell	Command to send	Behavior	Display
cd	CWD xxx	Change working remote directory	OK Changed to ‘new directory’
pwd	PWD	display current remote directory	OK Current directory
lcd	none	Change local directory	OK Local directory: ‘ …. ‘
ls	LIST	Return contents of remote directory	OK List contents of remote dir
get	RETR xxx	Return copy of file	OK GET: ‘ ‘
system	SYST	Return system type	OK System type
put	STOR xxx	Send a file to remote directory	OK PUT: ‘ ‘
bye	QUIT	Close connection	OK Connection closed
connect	none	Connect to remote host in active mode	OK Connection establised
exit	none	Exit the program	

IF error just
display the error codes
And description

This color is for Deliverables on Wednesday testing


ftpserver
This is the listening port
ftpclient.
ls = socket.socket()
ls.bind(ip, port)
ls.listen(5)
connect
cs = socket.socket()
cs.connect(ip, port)
Accept cause a new Socket to be created
cl_control_socket = ls.accept()
ls = socket.socket()
ls.bind(ip, port)
ls.listen(1)
Sendcmd('PORT')
recv PORT cmd
connect
cl_data_socket = socket.socket()
cl.data_socket.connect(ip, port)
Accept cause a new Socket to be created
ds = ls.accept()