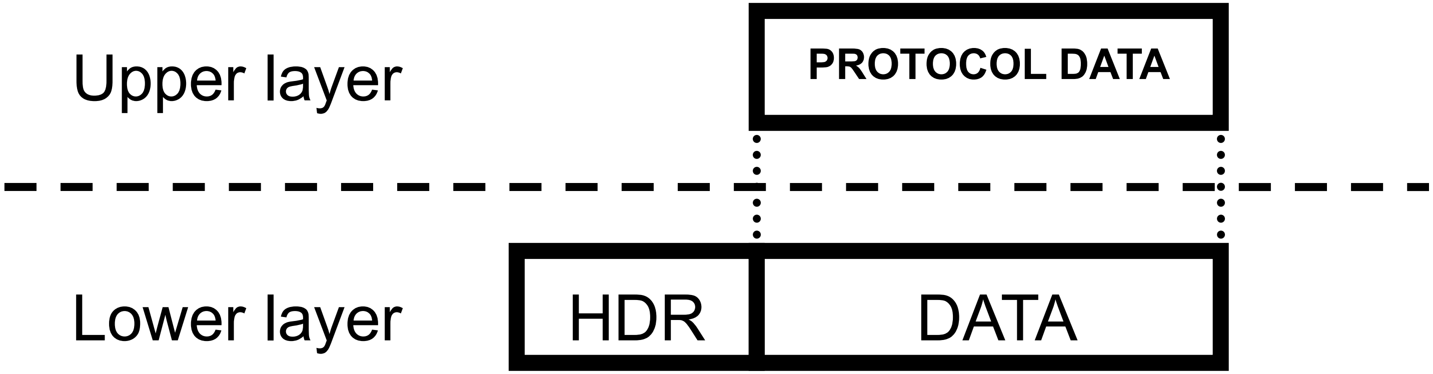
Network Programming in Python

Objectives

- Review principles of networking
- Contrast TCP and UDP features
- Show how Python programs access networking functionality
- Give examples of client and server program structures
- Demonstrate some Python network libraries
- Give pointers to other network functionality

Inter-Layer Relationships

- Each layer uses the layer below
  - The lower layer adds headers to the data from the upper layer
  - The data from the upper layer can also be a header on data from the layer above …



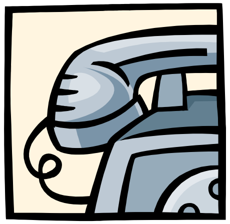
IP Characteristics

- Logical (32-bit) addresses
  - Unrelated to physical addressing
  - Leading bits determine network membership
- Datagram-based
  - Connectionless
- Unreliable
  - Best effort delivery
  - No delivery guarantees

UDP Characteristics

- Also datagram-based
  - Connectionless, unreliable, can broadcast
- Applications usually message-based
  - No transport-layer retries
  - Applications handle (or ignore) errors
- Processes identified by port number
- Services live at specific ports
  - Usually below 1024, requiring privilege

TCP Characteristics

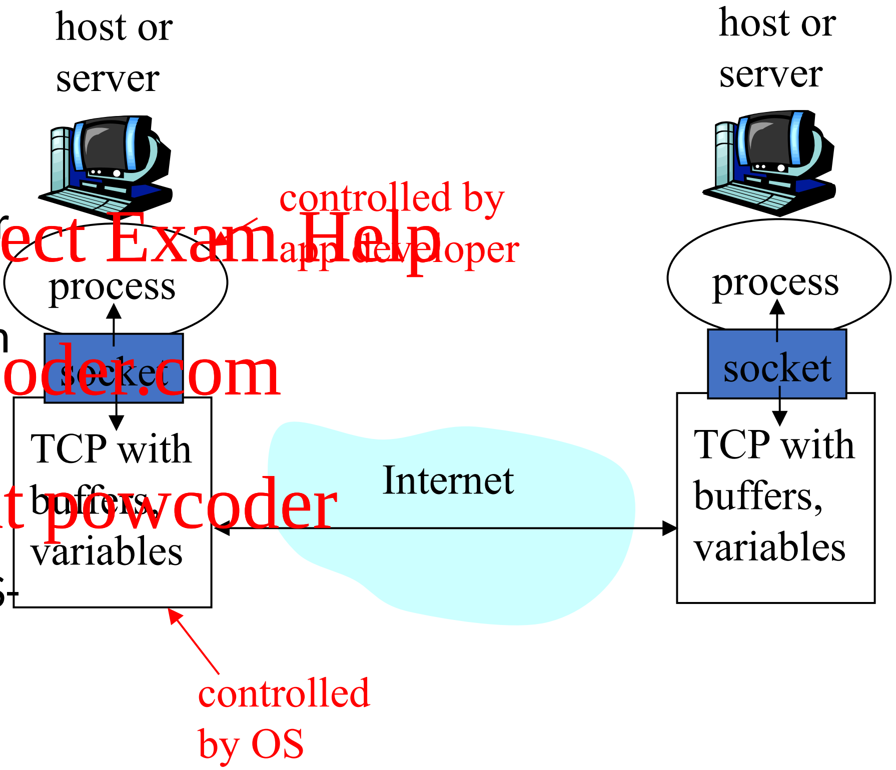


- Connection-oriented
  - Two endpoints of a virtual circuit
- Reliable
  - Application needs no error checking
- Stream-based
  - No predefined blocksize
- Processes identified by port numbers
- Services live at specific ports

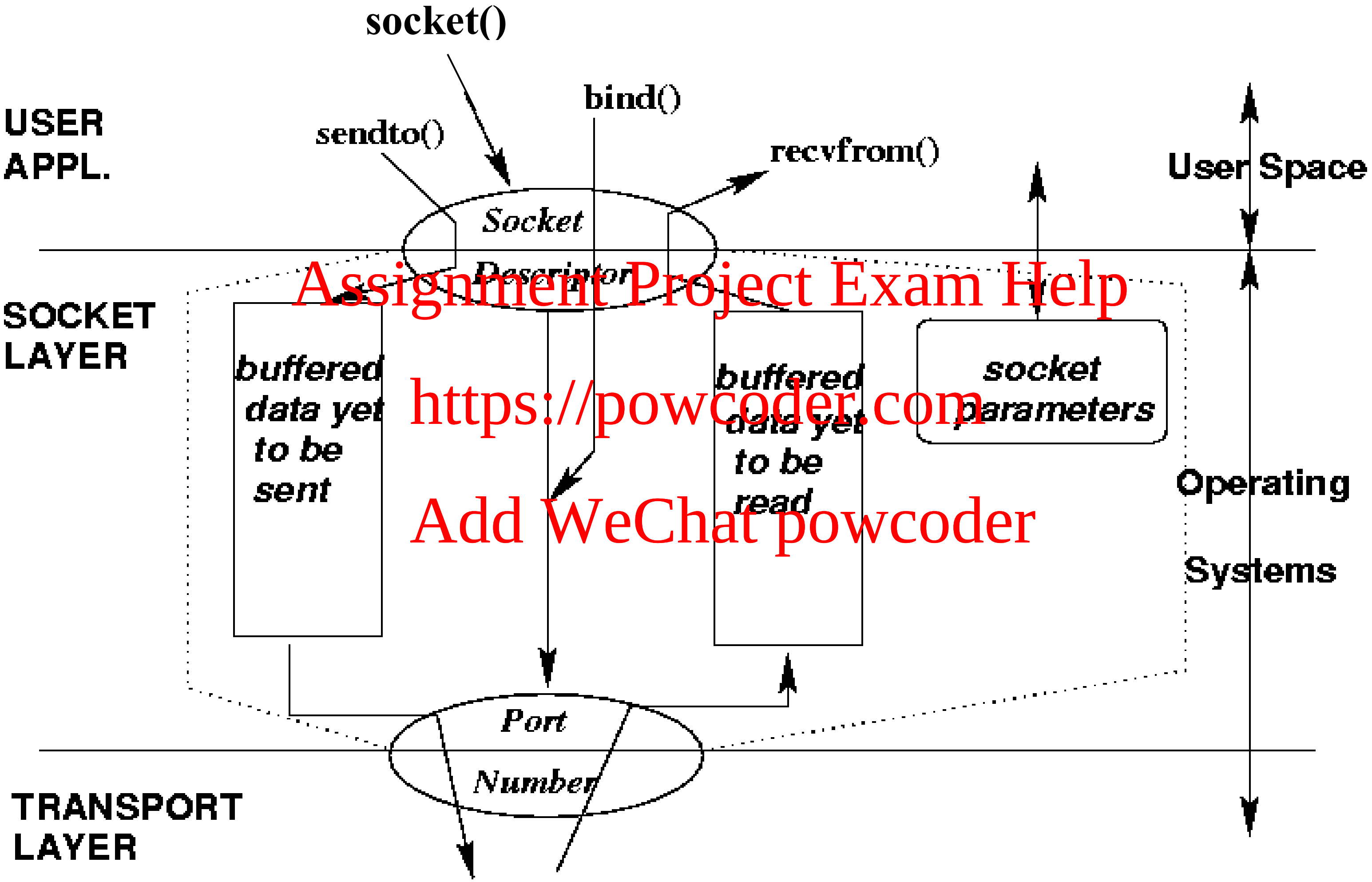
Socket Programming API

Application Programming Interface

- *Socket* analogous to door
  - sending process shoves message out door
  - sending process assumes transport infrastructure on other side of door which brings message to socket at receiving process
  - host-local, application created/owned, OS-controlled
  - connection between sockets set-up/ managed by OS


host or server
host or server
controlled by app developer
process
process
socket
socket
TCP with buffers, variables
Internet
TCP with buffers, variables
controlled by OS

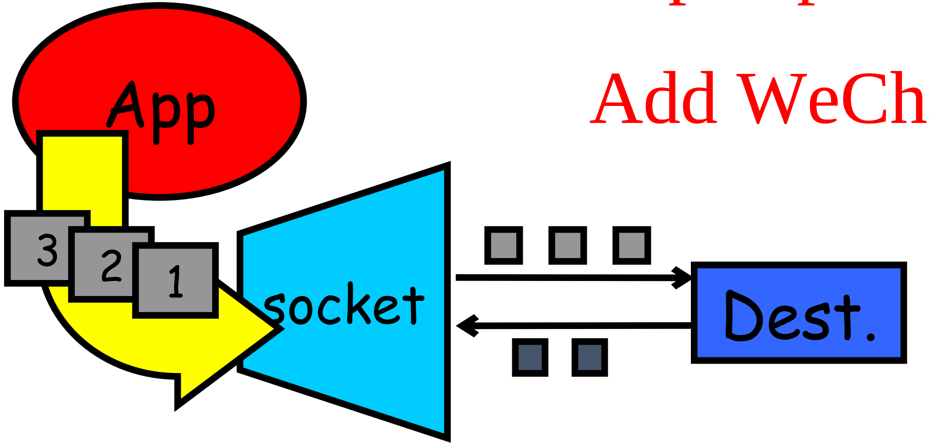
Socket: Conceptual View

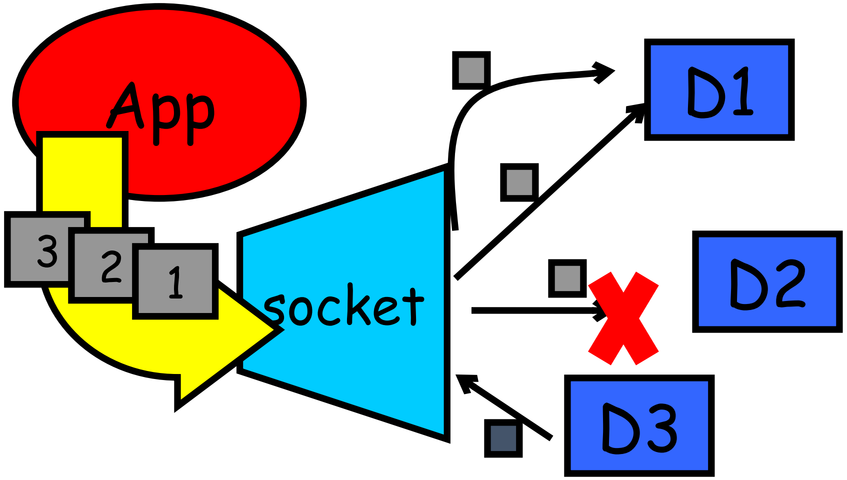


Two essential types of sockets

- SOCK_STREAM
  - a.k.a. TCP
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional

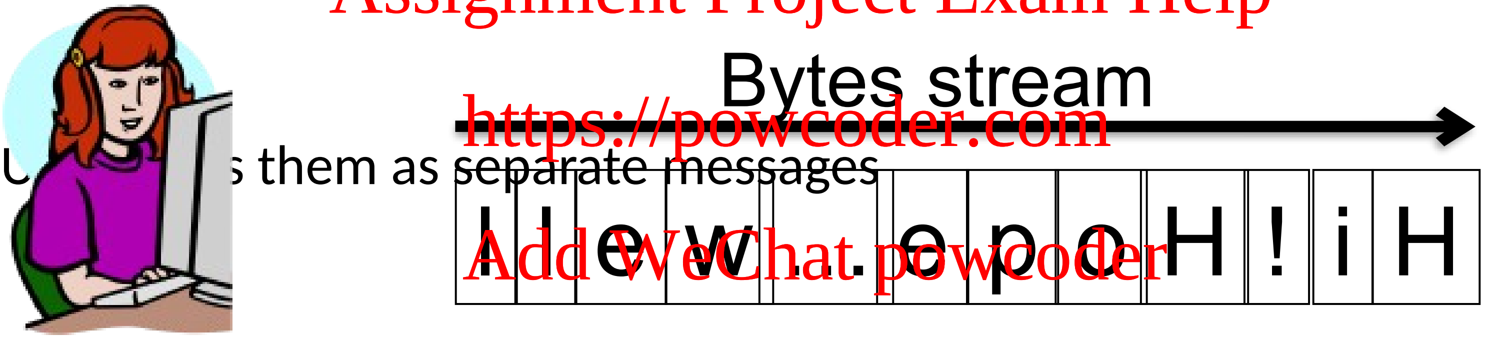
- SOCK_DGRAM
  - a.k.a. UDP
  - unreliable delivery
  - no order guarantees
  - no notion of “connection” – app indicates dest. for each packet
  - can send or receive



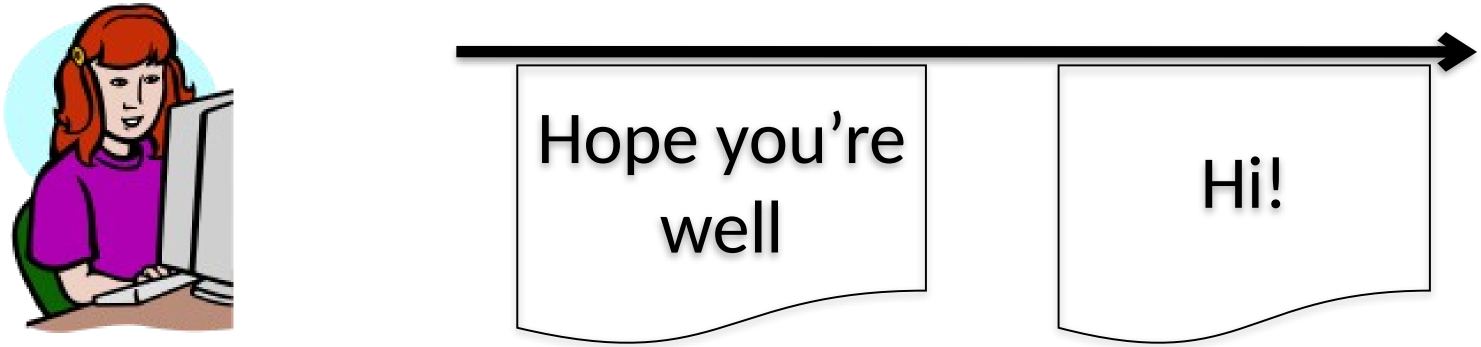


Types of Sockets

- When sending “Hi!” and “Hope you’re well”
- TCP treats them as a single bytes stream

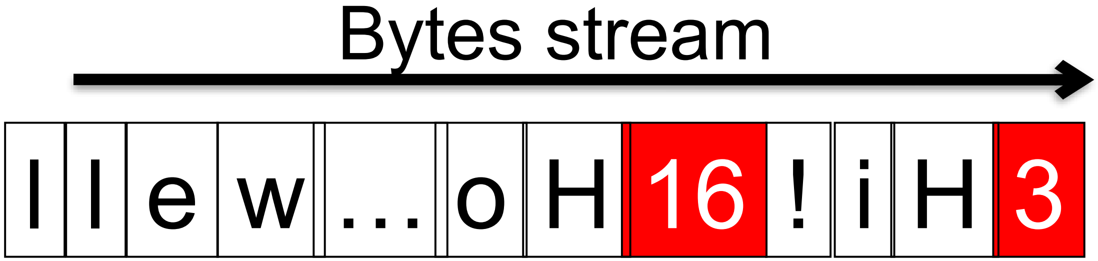


- U[illegible]s them as separate messages



Types of Sockets (cont'd)

- Thus, TCP needs application-level message boundary.
  - By carrying length in application-level header

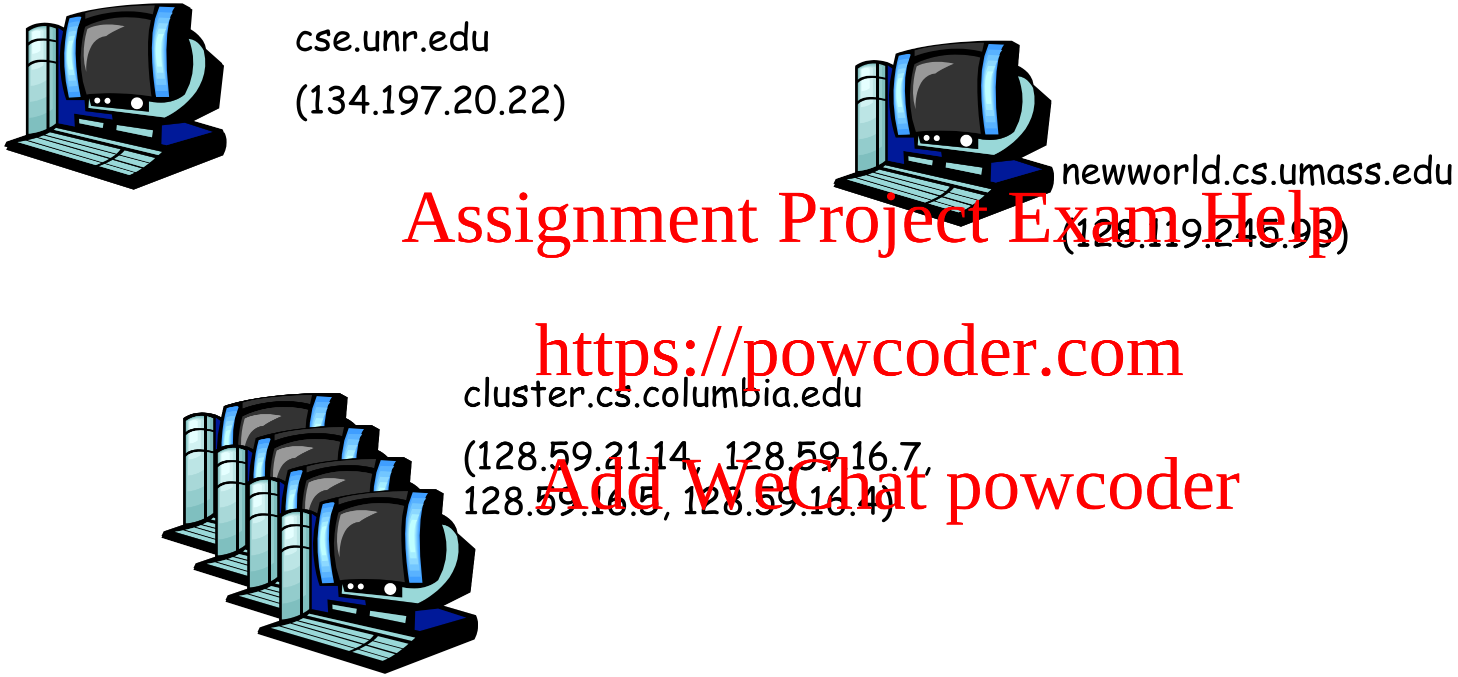


l	l	e	w	…	o	H	16	!	i	H	3

Client/Server Concepts

- Server opens a specific port
  - The one associated with its service
  - Then just waits for requests
  - Server is the passive opener
- Clients get ephemeral ports
  - Guaranteed unique, 1024 or greater
  - Uses them to communicate with server
  - Client is the active opener

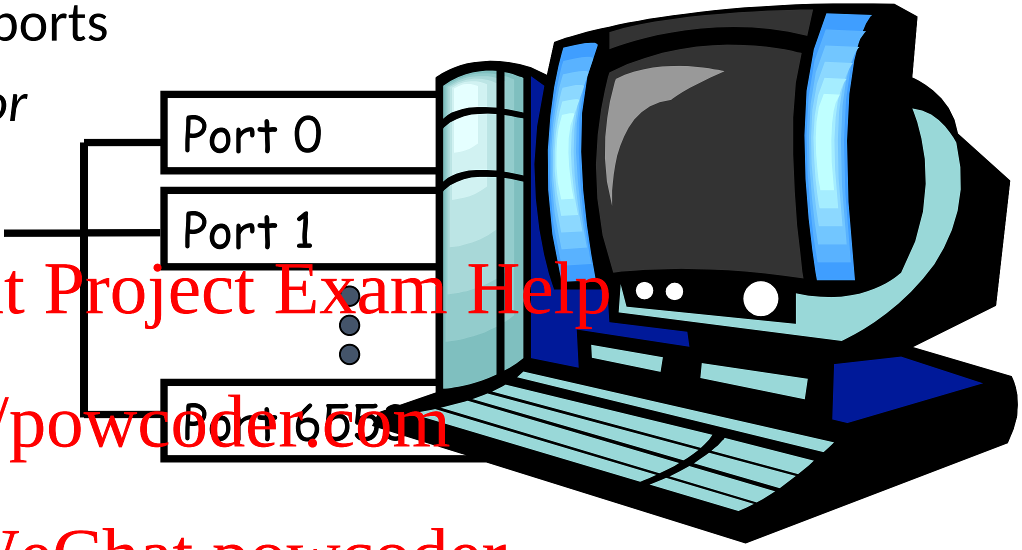
A Socket-eye view of the Internet



- Each host machine has an IP address

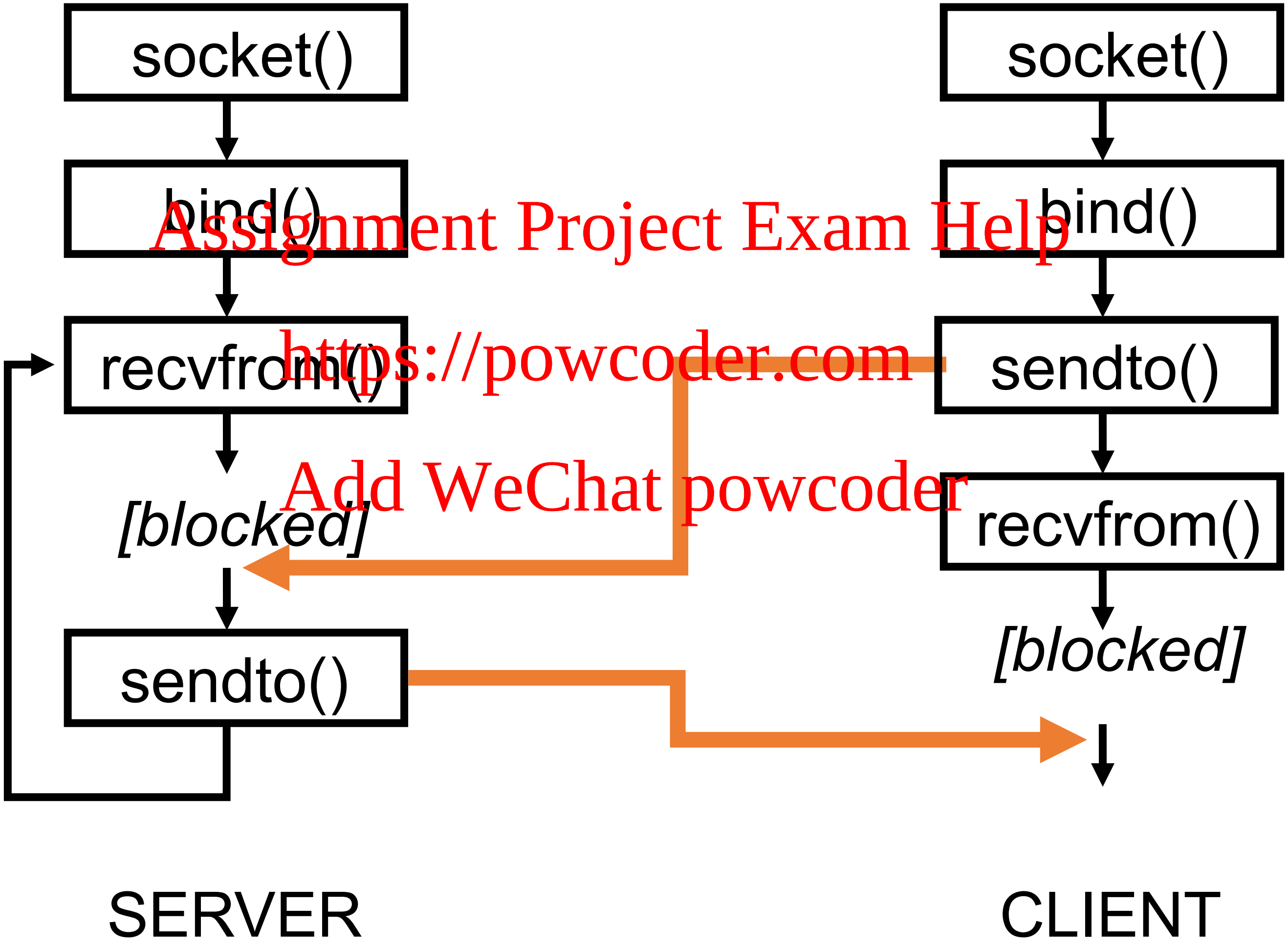
Ports

- Each host has 65,536 ports
- Some ports are *reserved for specific apps*
  - 20,21: FTP
  - 23: Telnet
  - 80: HTTP
  - see RFC 1700
    - about 2000 ports are reserved



A socket provides an interface to send data to/from the network through a port

Connectionless Services



Simple Connectionless Server

```
from socket import socket, AF_INET, SOCK_DGRAM
s = socket(AF_INET, SOCK_DGRAM)
s.bind(('127.0.0.1', 11111))
while True:
    data, addr = s.recvfrom(1024)
    print "Connection from", addr
    s.sendto(data.upper(), addr)
```

Empty -> all

Note that the *bind()* argument is a two-element tuple of address and port number

See: socket & select from https://docs.python.org/2.4/lib/someos.html

Simple Connectionless Client

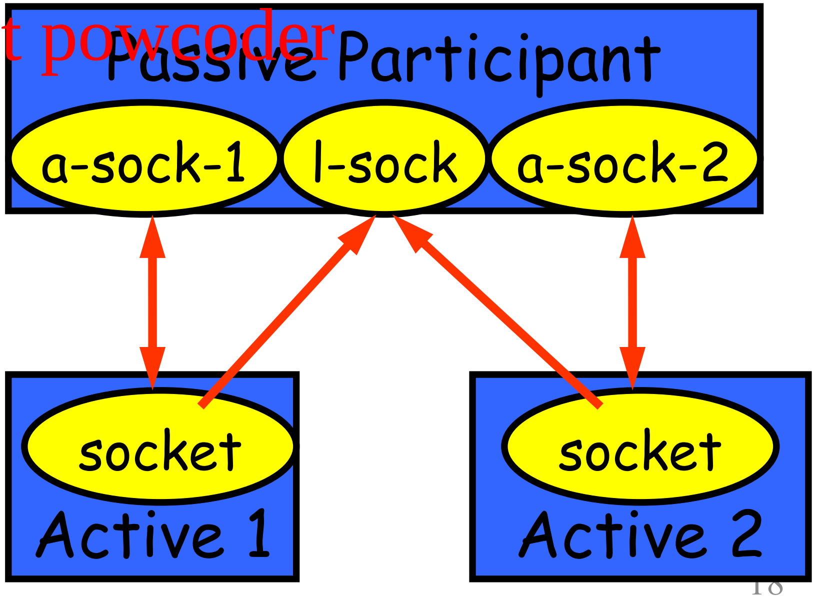
```
from socket import socket, AF_INET, SOCK_DGRAM
s = socket(AF_INET, SOCK_DGRAM)
s.bind(('127.0.0.1', 0)) # OS chooses port
print "using", s.getsocketname()
server = ('127.0.0.1', 11111)
s.sendto("MixedCaseString", server)
data, addr = s.recvfrom(1024)
print "received", data, "from", addr
s.close()
```

Connection setup cont'd

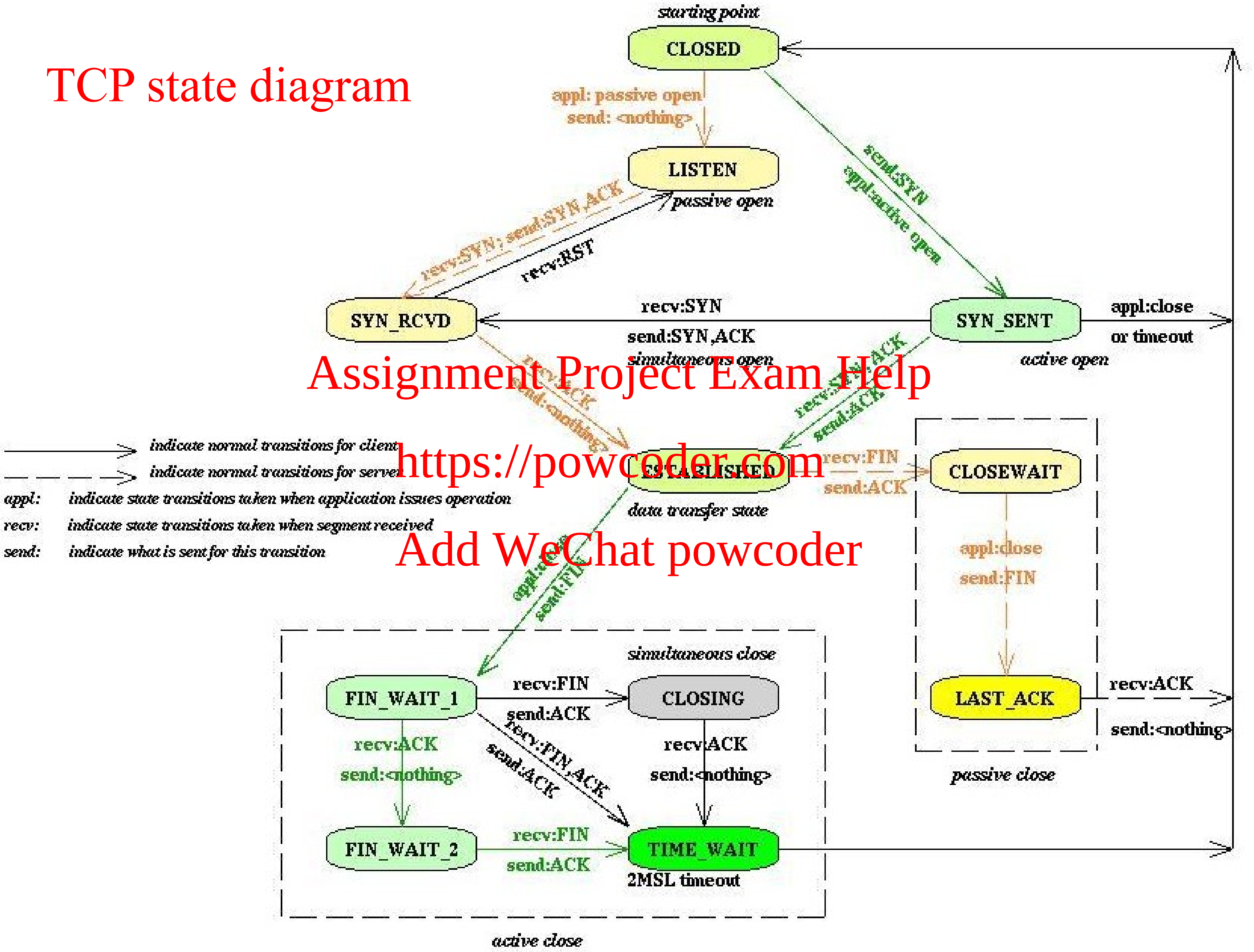
- Passive participant
  - step 1: listen (for incoming requests)
  - step 3: accept (a request)
  - step 4: data transfer

- Active participant
  - step 2: request & establish connection
  - step 4: data transfer

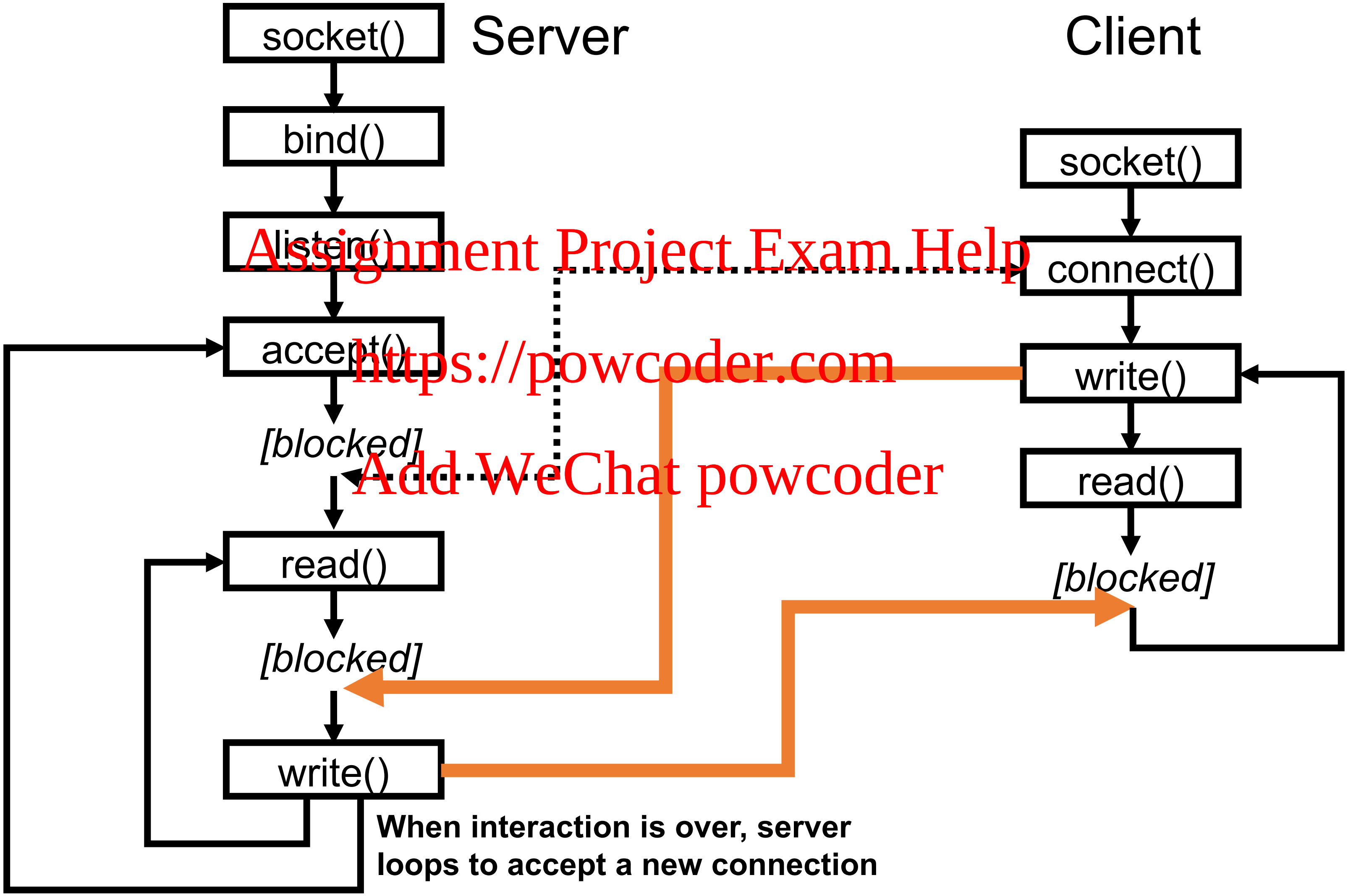
- The accepted connection is on a new socket
- The old socket continues to listen for other active participants



TCP state diagram



Connection-Oriented Services



When interaction is over, server loops to accept a new connection

Connection-Oriented Server

```
from socket import socket, AF_INET,
SOCK_STREAM
s = socket(AF_INET, SOCK_STREAM)
s.bind(('127.0.0.1', 9999))
s.listen(5) # max queued connections
while True:
    sock, addr = s.accept()
    # use socket sock to communicate
    # with client process
```

- Client connection creates new socket
    - Returned with address by ***accept()***
- Server handles one client at a time

Connection-Oriented Client

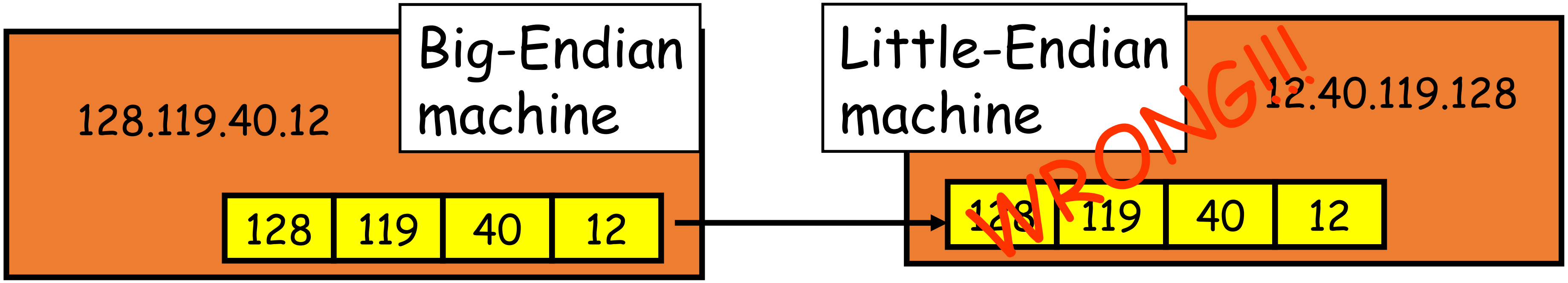
```
from socket import socket, AF_INET,
 SOCK_STREAM
(SERVER, PORT) = ('127.0.0.1', 9999)
s = socket(AF_INET, SOCK_STREAM)
s.connect((SERVER, PORT))
s.send('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', data
```

This is a simple example

- Sends message, receives response
- Server receives 0 bytes after ***close()***

Some socket Utility Functions

- **`htonl(i), htons(i)`**
  - 32-bit or 16-bit integer to network format
- **`ntohl(i), ntohs(i)`**
  - 32-bit or 16-bit integer to host format
- **`inet_aton(ipstr), inet_ntoa(packed)`**
  - Convert addresses between regular strings and 4-byte packed strings



Handling Names & Addresses

- **`getfqdn(host='')`**
  - Get canonical host name for host

- **`gethostbyaddr(ipaddr)`**
- **`gethostbyname_ex(hostname)`**
  - Returns (hostname, aliases, addresses)
    - Hostname is canonical name
    - Aliases is a list of other names
    - Addresses is a list of IP address strings

Treating Sockets as Files

- **`makefile([mode[, bufsize]])`**
  - Creates a file object that references the socket
  - Makes it easier to program to handle data streams
    - No need to assemble stream from buffers

Summary of Address Families

- **`socket.AF_UNIX`**
  - Unix named pipe (NOT Windows…)
- **`socket.AF_INET`**
  - Internet – IP version 4
  - The basis of this class
- **`socket.AF_INET6`**
  - Internet – IP version 6
  - Rather more complicated …

Summary of Socket Types

- `socket.SOCK_STREAM`
  - TCP, connection-oriented
- `socket.SOCK_DGRAM`
  - UDP, connectionless
- `socket.SOCK_RAW`
  - Gives access to subnetwork layer
- `SOCK_RDM, SOCK_SEQPACKET`
  - Very rarely used

Timeout Capabilities

- Can set a default for all sockets
  - **`socket.setdefaulttimeout(seconds)`**
  - Argument is float # of seconds
  - Or **`None`** (indicates no timeout)

- Can set a timeout on an existing socket **`s`**
  - **`s.settimeout(seconds)`**

Server Libraries

- **`SocketServer`** module provides basic server features
- Subclass the **`TCPServer`** and **`UDPServer`** classes to serve specific protocols
- Subclass **`BaseRequestHandler`**, overriding its handle() method, to handle requests
- Mix-in classes allow asynchronous handling via **`ThreadingMixIn`**

Using SocketServer Module

- Server instance created with address and handler-class as arguments:
  `SocketServer.UDPServer(myaddr, MyHandler)`
- Each connection/transmission creates a request handler instance by calling the handler-class*
- Created handler instance handles a message (UDP) or a complete client session (TCP)

* In Python you instantiate a class by calling it like a function