Python Object-Oriented Program with Libraries

Unit 3: Web Programming

CHAPTER 1: NETWORK FUNDAMENTALS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Python Networking Overview

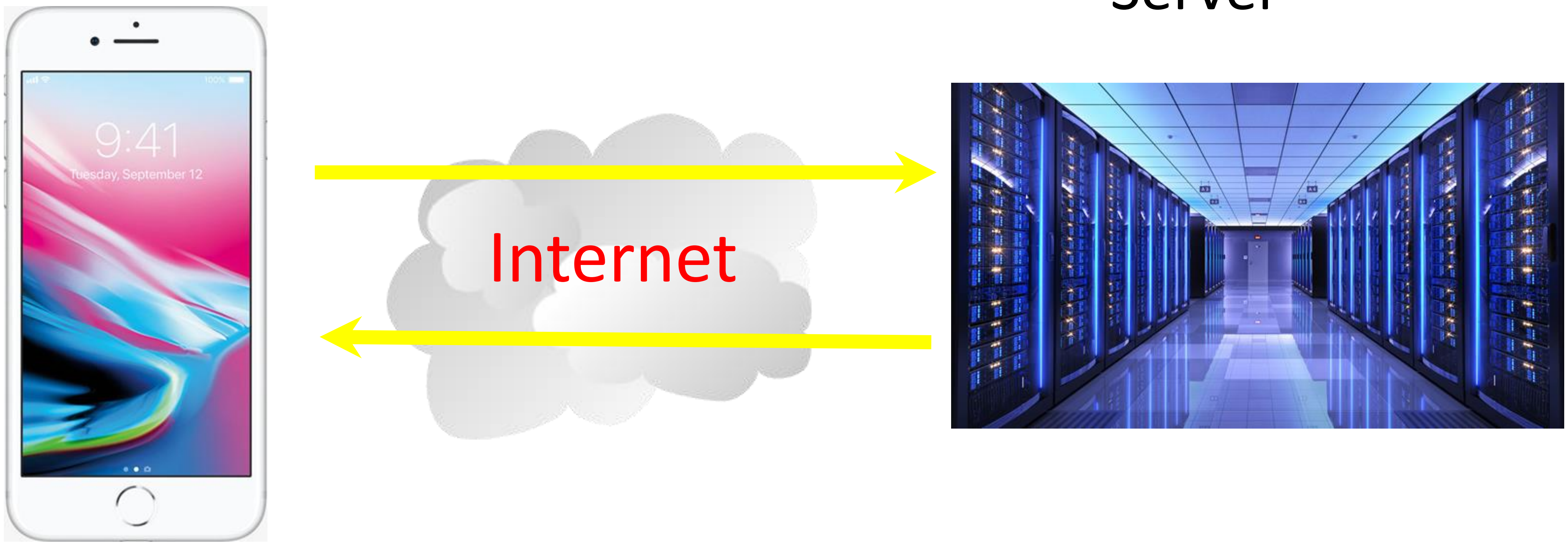
LECTURE 1

Python Networking

- 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
- Writing network programs in Python tends to be substantially easier than in C/C++

This Unit

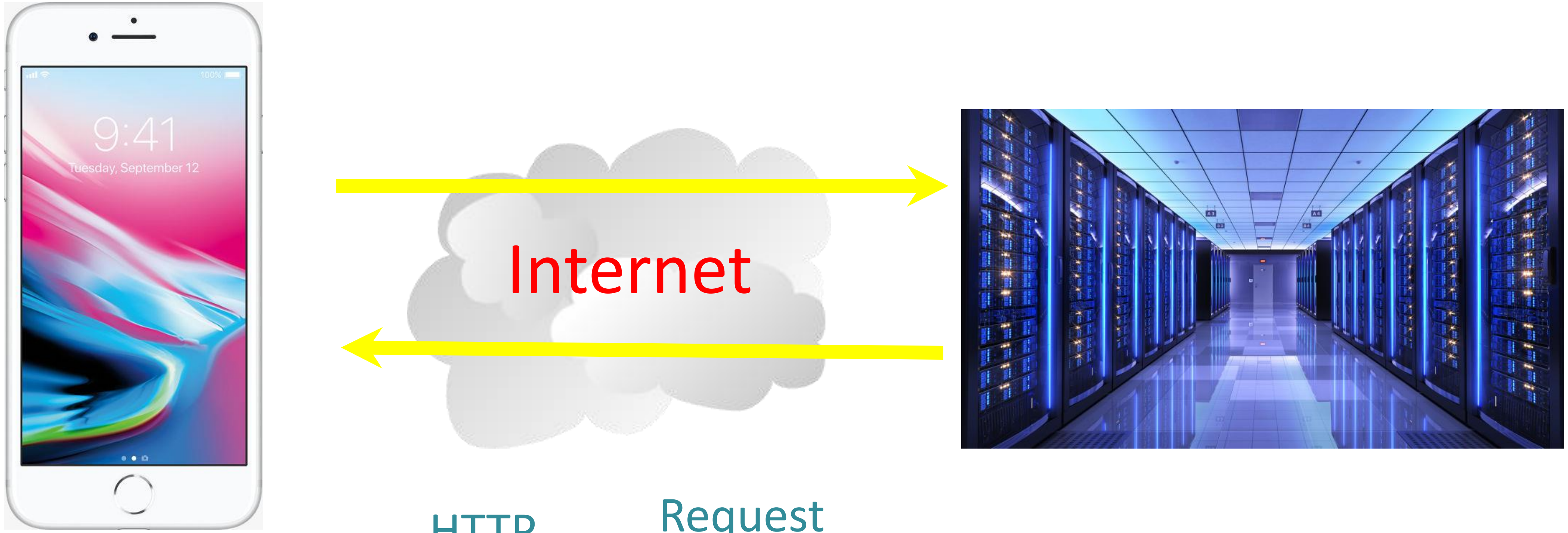
- This course focuses on the essential details of network programming that all Python programmers should probably know
  1. **Sockets:** Low-level programming with sockets
  2. **Client:** High-level client modules
  3. **Data:** How to deal with common data encodings
  4. **Protocol:** Simple web programming (HTTP)
  5. **Parallelism:** Simple distributed computing



Client

Server

Internet



HTML

JavaScript

AJAX

CSS

HTTP

Request

Response

GET

socket

POST

Python

Data Store

Templates

memcache

Standard Library

- We will only cover modules supported by the Python standard library
- These come with Python by default
- Keep in mind, much more functionality can be found in third-party modules
- Will give links to notable third-party libraries as appropriate

Prerequisites

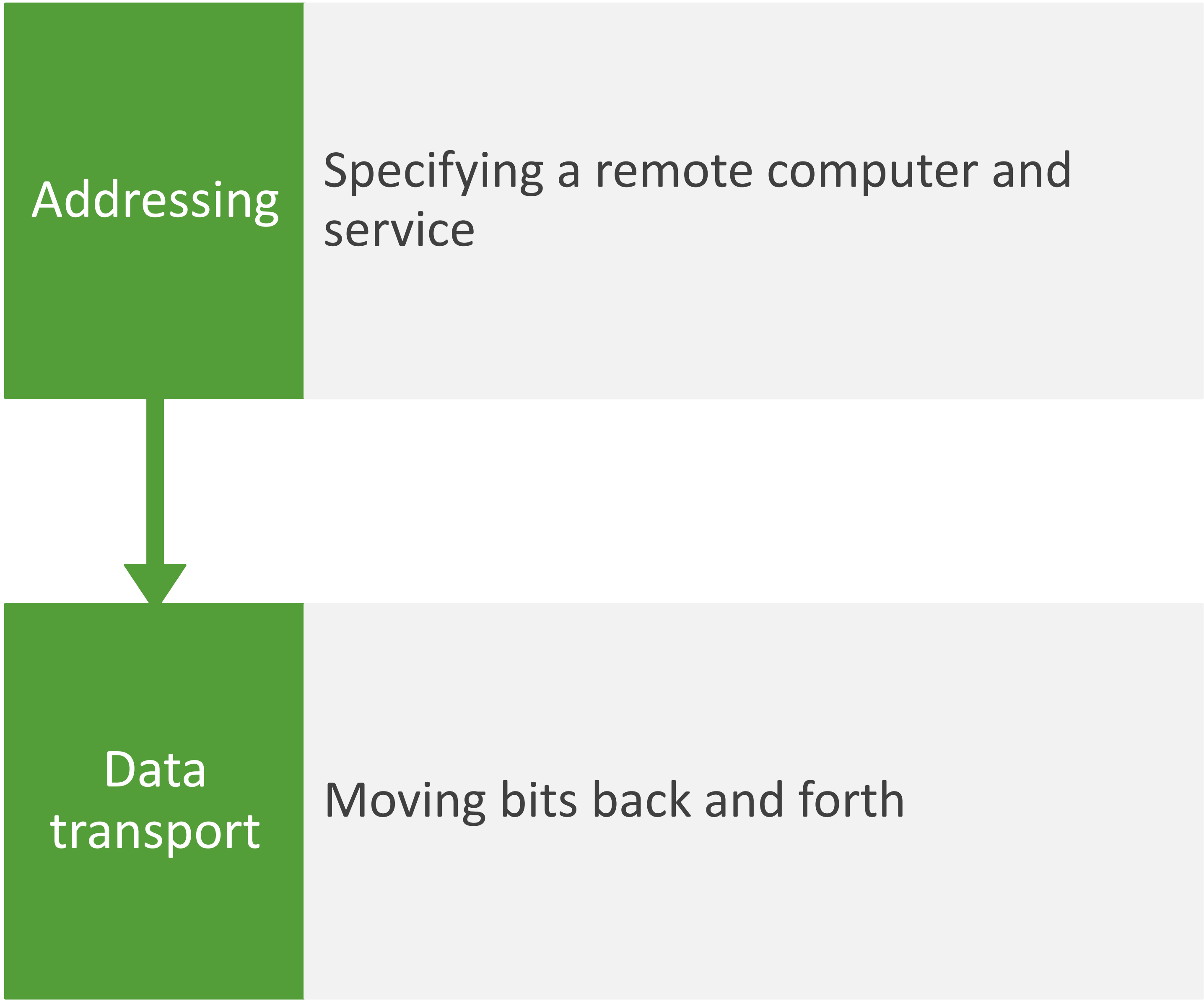
- You should already know Python basics
- However, you don't need to be an expert on all of its advanced features (in fact, none of the code to be written is highly sophisticated)
- You should have some prior knowledge of systems programming and network concepts

Network Fundamentals

LECTURE 2

The Problem
It's just sending/receiving bits.

Two Main Issues



Addressing

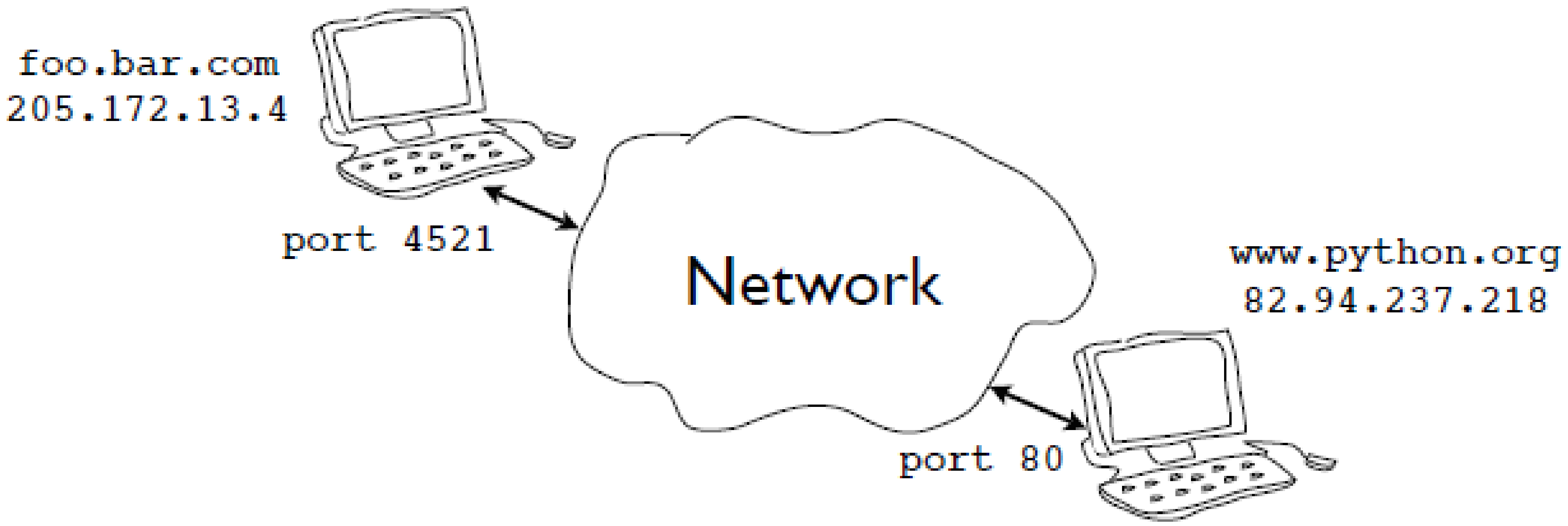
Specifying a remote computer and service

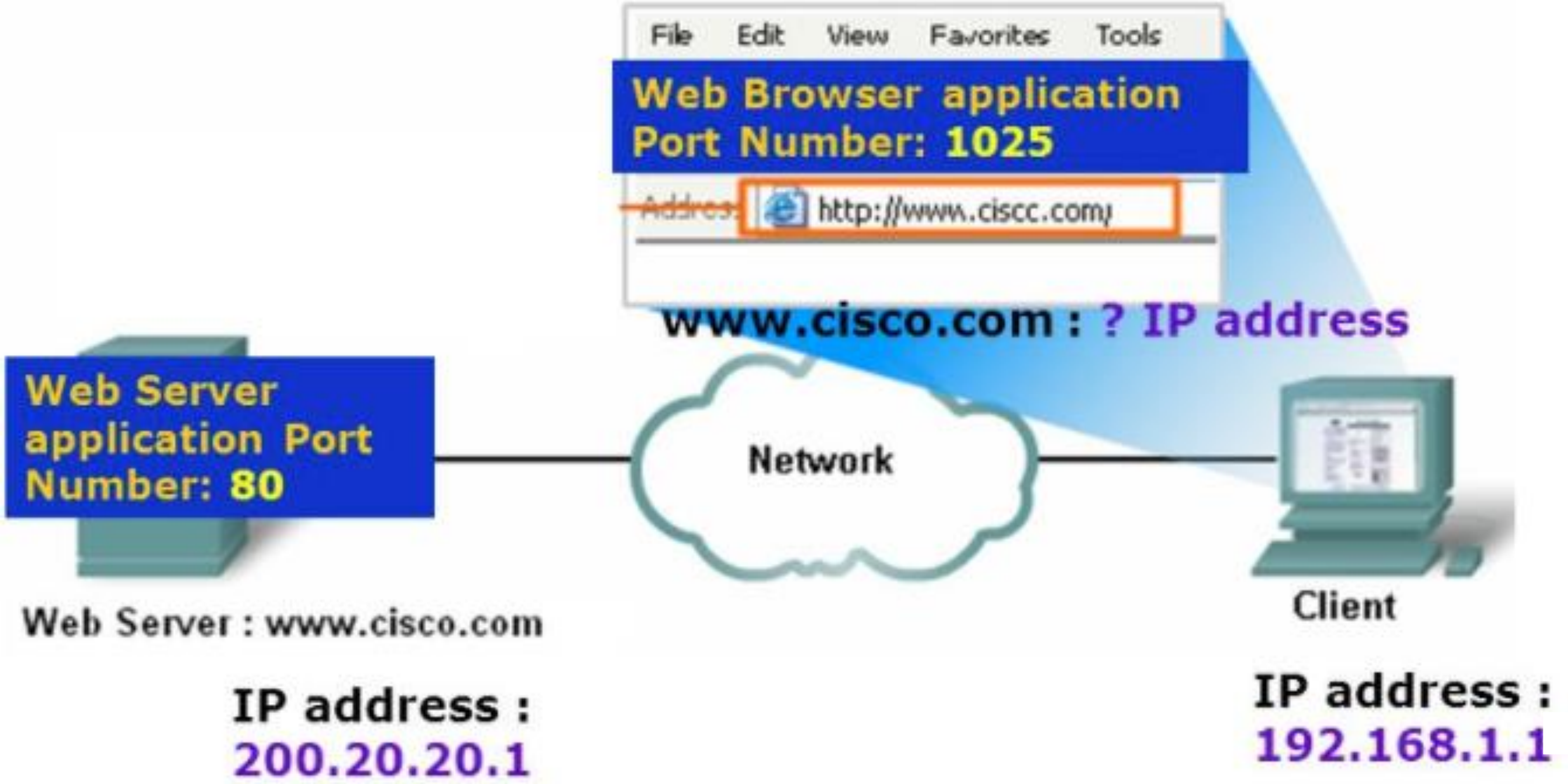
Data transport

Moving bits back and forth

Network Addressing

- Machines have a hostname and IP address
- Programs/services have port numbers





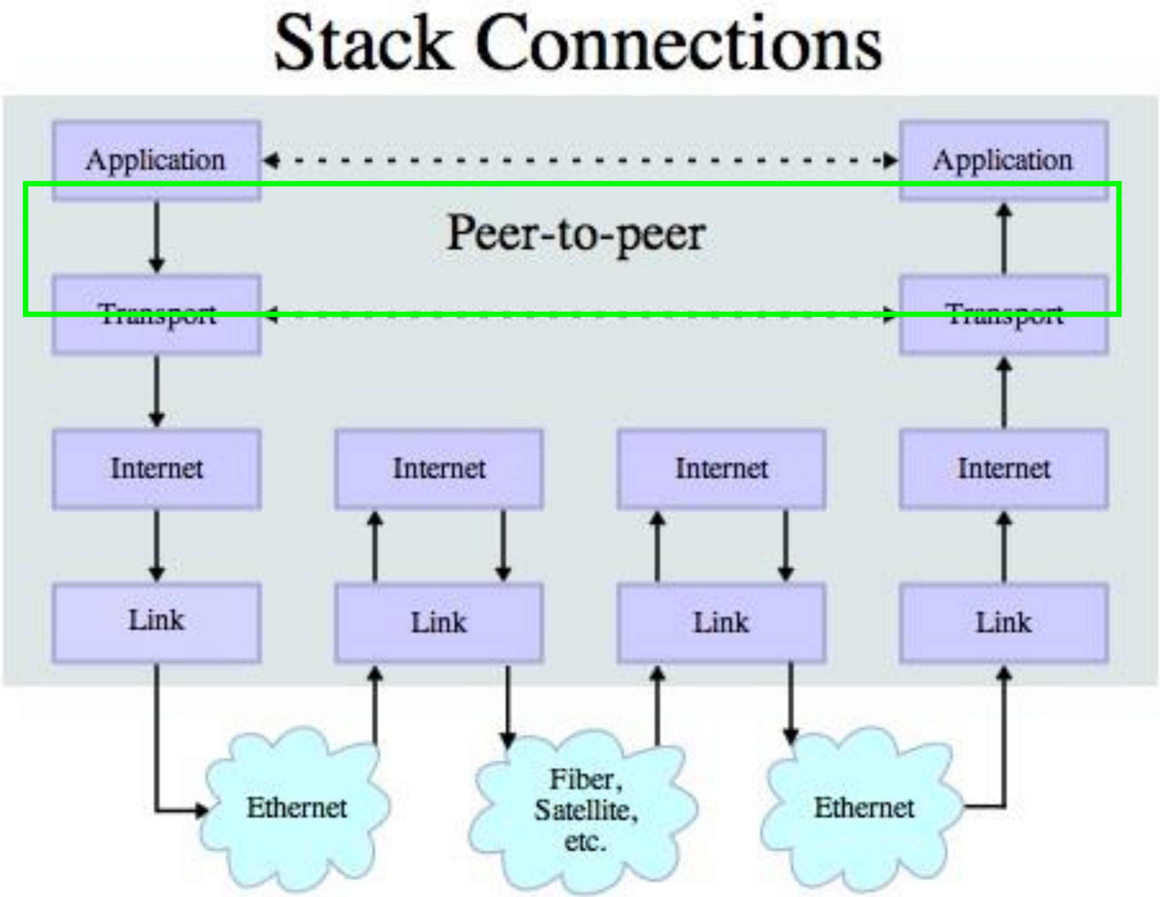
Domain Name, IP Address and Port Number

Network Architecture

LECTURE 3

Transport Control Protocol (TCP)

- Built on top of IP (Internet Protocol)
- Assumes IP might lose some data - stores and retransmits data if it seems to be lost
- Handles “flow control” using a transmit window
- Provides a nice reliable pipe

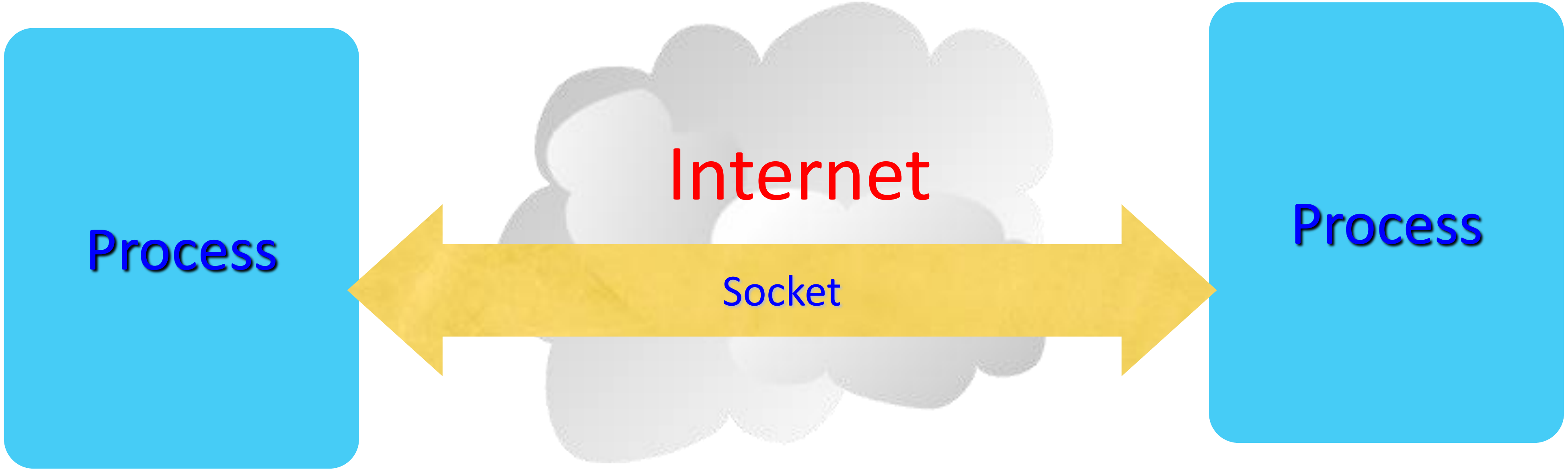


Source: http://en.wikipedia.org/wiki/Internet_Protocol_Suite

TCP Connections / Sockets

Process to Process Communication

"In computer networking, an Internet socket or network socket is an endpoint of a bidirectional inter-process communication flow across an Internet Protocol-based computer network, such as the Internet."



http://en.wikipedia.org/wiki/Internet_socket

TCP Port Numbers

- A port is an application-specific or process-specific software communications endpoint
- It allows multiple networked applications to coexist on the same server.
- There is a list of well-known TCP port numbers

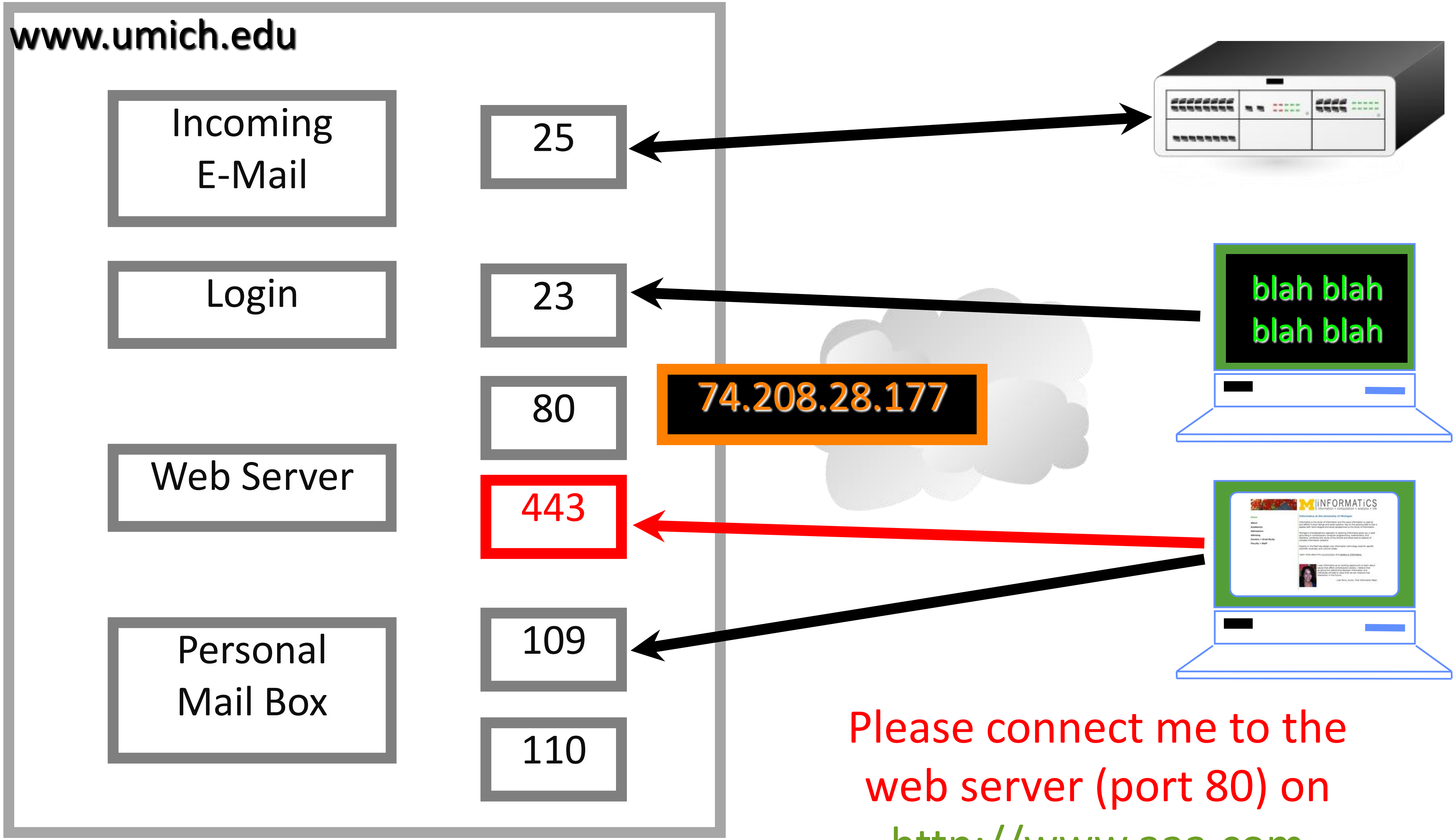
http://en.wikipedia.org/wiki/TCP_and_UDP_port

Standard Ports

Ports for common services are preassigned:

- 21 **FTP**
- 22 **SSH**
- 23 **Telnet**
- 25 **SMTP** (Mail)
- 53 **DNS** (Domain Name)
- 80 **HTTP** (Web)   - 443 **HTTPS** (web, Secure)
- 110 **POP3** (Mail) - 119 **NNTP** (News)
- (143/220/993) **IMAP** - Mail Retrieval

- Other port numbers may just be randomly assigned to programs by the operating system



www.umich.edu

Incoming E-Mail

25

Login

23

80

Web Server

443

Personal Mail Box

109

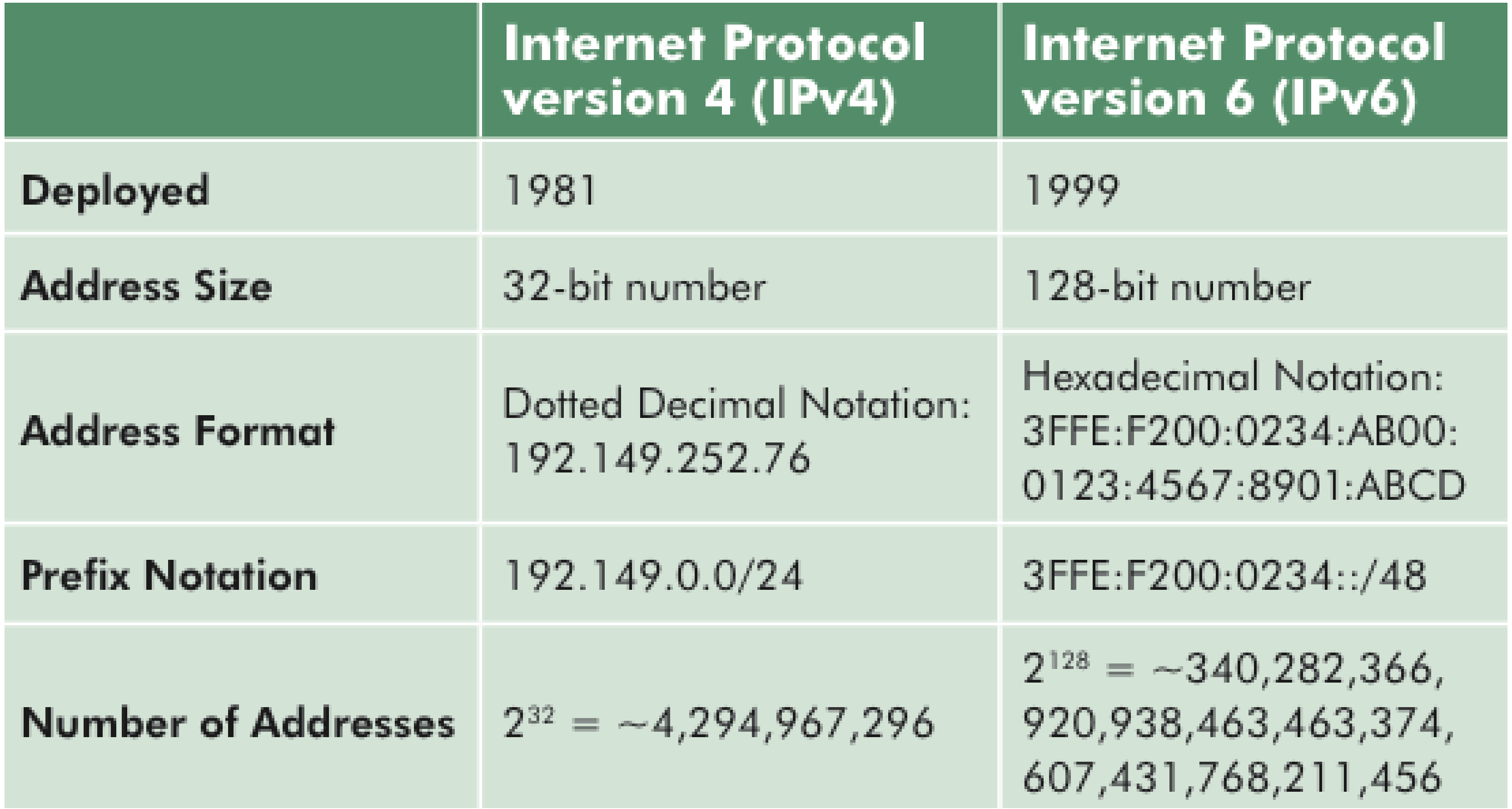
110

74.208.28.177

Please connect me to the web server (port 80) on http://www.aaa.com

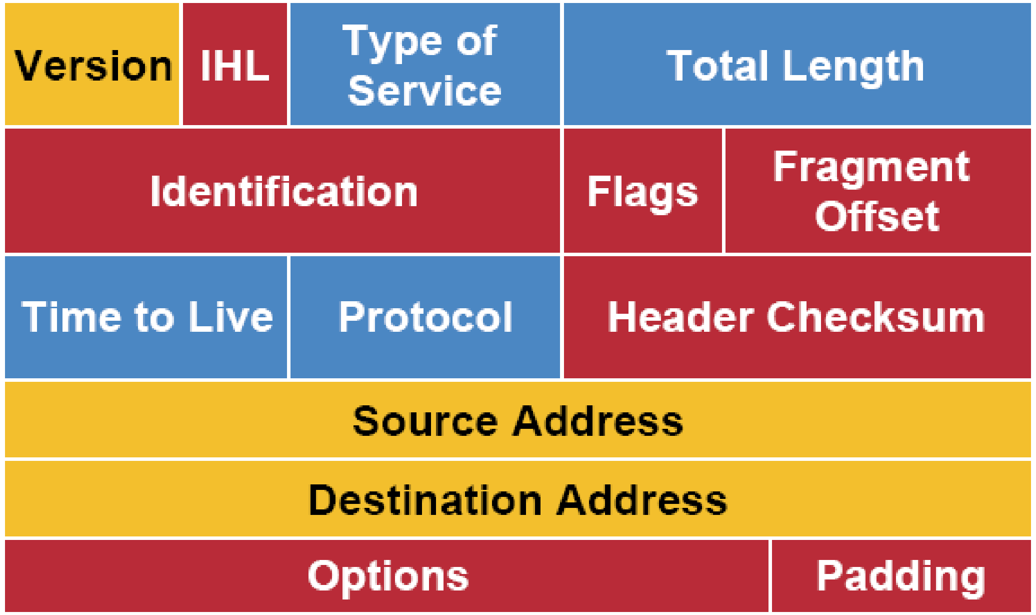
http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

IP Address



	Internet Protocol version 4 (IPv4)	Internet Protocol version 6 (IPv6)
Deployed	1981	1999
Address Size	32-bit number	128-bit number
Address Format	Dotted Decimal Notation: 192.149.252.76	Hexadecimal Notation: 3FFE:F200:0234:AB00: 0123:4567:8901:ABCD
Prefix Notation	192.149.0.0/24	3FFE:F200:0234::/48
Number of Addresses	2^{32} = ~4,294,967,296	2^{128} = ~340,282,366, 920,938,463,463,374, 607,431,768,211,456

IPv4 Header



Version | IHL | Type of Service | Total Length

Identification | Flags | Fragment Offset

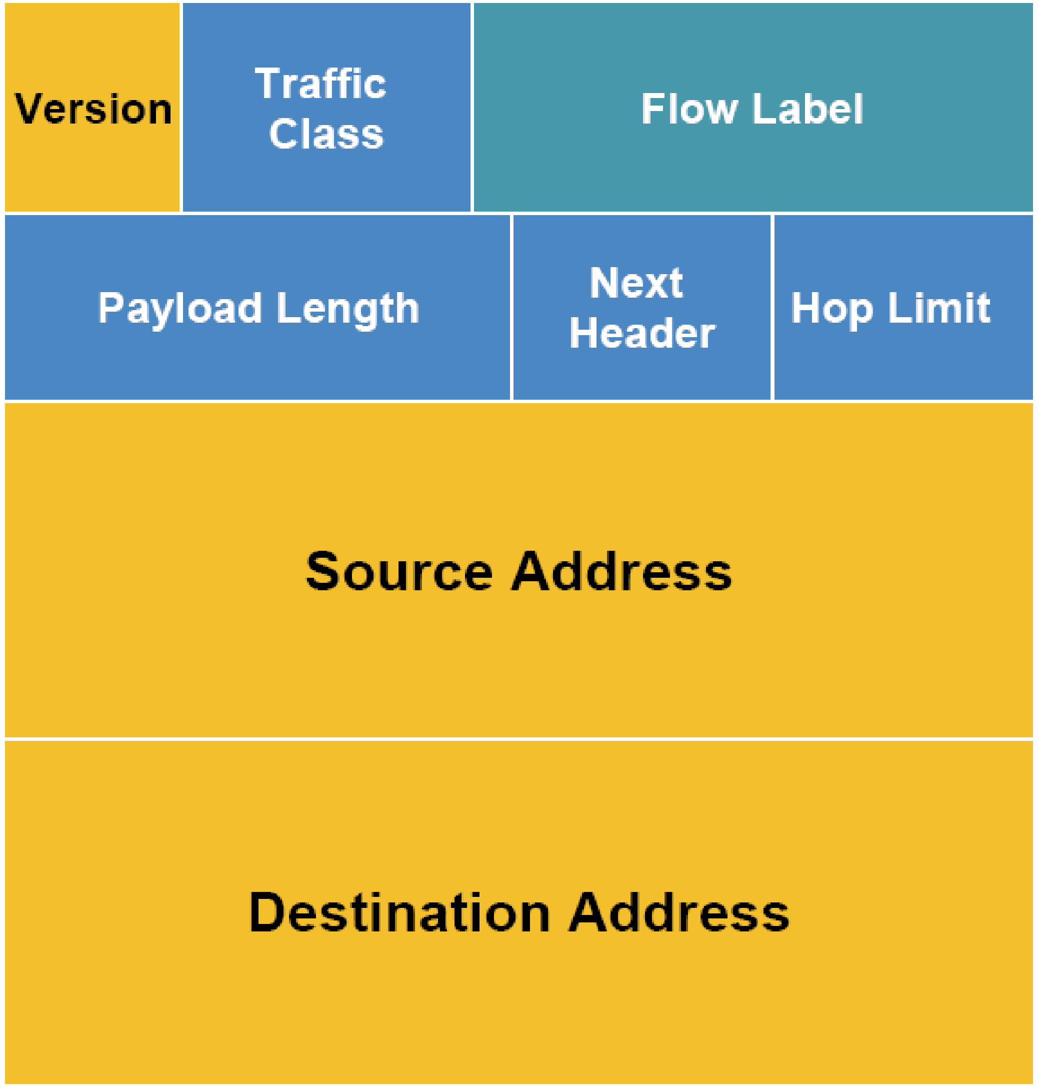
Time to Live | Protocol | Header Checksum

Source Address

Destination Address

Options | Padding

IPv6 Header



Version | Traffic Class | Flow Label

Payload Length | Next Header | Hop Limit

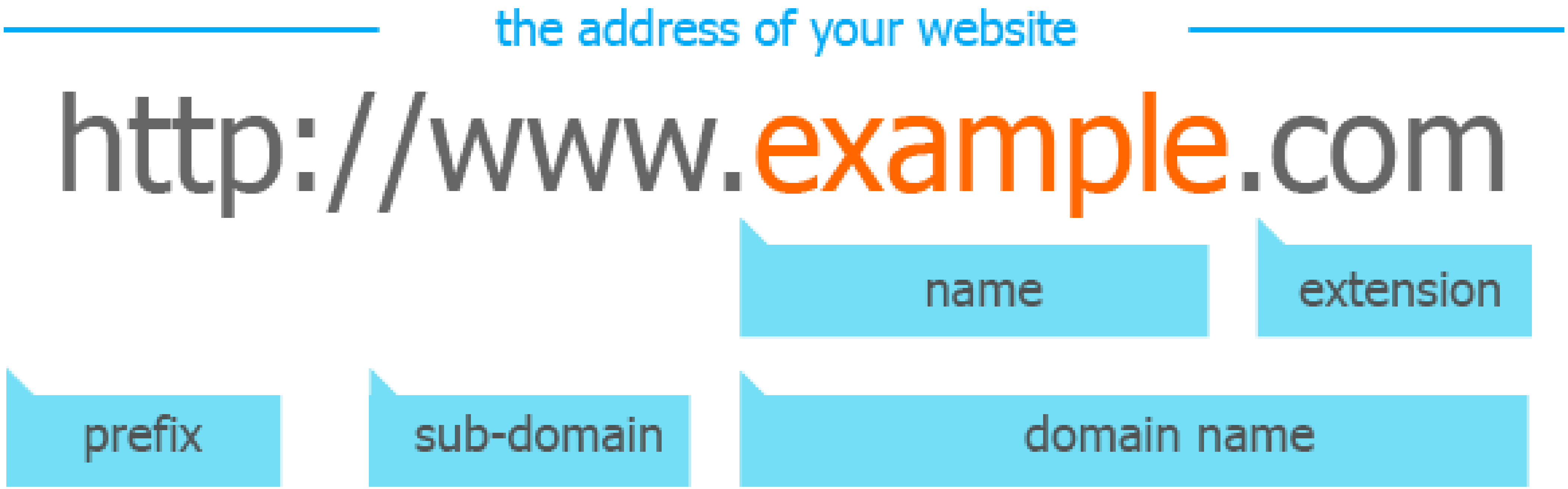
Source Address

Destination Address

Legend

- 🟨 Field’s Name Kept from IPv4 to IPv6
- 🟥 Fields Not Kept in IPv6
- 🟦 Name and Position Changed in IPv6
- 🟩 New Field in IPv6

Domain Name System (DNS)



the address of your website

http://www.example.com

name

extension

prefix

sub-domain

domain name

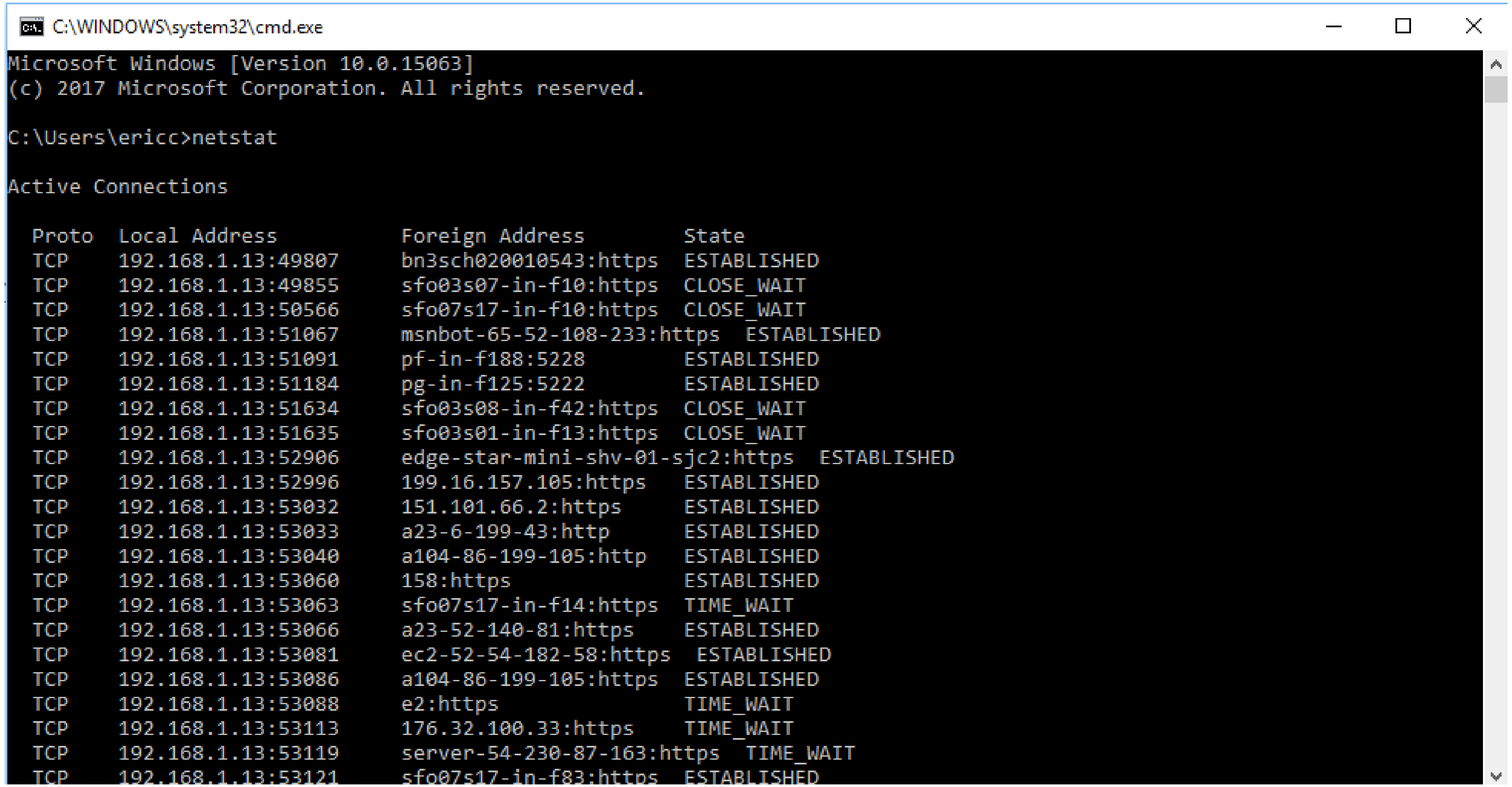
PuTTY (Terminal)

LECTURE 4

PuTTY

- **PuTTY** is a free implementation of **SSH** and **Telnet** for Windows and Unix platforms, along with an **xterm** terminal emulator. It is written and maintained primarily by Simon Tatham.

- **SSH**: Secure Shell (**SSH**) is a cryptographic network protocol for operating network services securely over an unsecured network
- **Telnet**:a network protocol that allows a user on one computer to log onto another computer that is part of the same network.
- **TTY**: virtual terminal



```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\ericc>netstat

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    192.168.1.13:49807     bn3sch020010543:https  ESTABLISHED
  TCP    192.168.1.13:49855     sfo03s07-in-f10:https  CLOSE_WAIT
  TCP    192.168.1.13:50566     sfo07s17-in-f10:https  CLOSE_WAIT
  TCP    192.168.1.13:51067     msnbot-65-52-108-233:https  ESTABLISHED
  TCP    192.168.1.13:51091     pf-in-f188:5228        ESTABLISHED
  TCP    192.168.1.13:51184     pg-in-f125:5222        ESTABLISHED
  TCP    192.168.1.13:51634     sfo03s08-in-f42:https  CLOSE_WAIT
  TCP    192.168.1.13:51635     sfo03s01-in-f13:https  CLOSE_WAIT
  TCP    192.168.1.13:52906     edge-star-mini-shv-01-sjc2:https  ESTABLISHED
  TCP    192.168.1.13:52996     199.16.157.105:https   ESTABLISHED
  TCP    192.168.1.13:53032     151.101.66.2:https     ESTABLISHED
  TCP    192.168.1.13:53033     a23-6-199-43:http      ESTABLISHED
  TCP    192.168.1.13:53040     a104-86-199-105:http   ESTABLISHED
  TCP    192.168.1.13:53060     158:https              ESTABLISHED
  TCP    192.168.1.13:53063     sfo07s17-in-f14:https  TIME_WAIT
  TCP    192.168.1.13:53066     a23-52-140-81:https    ESTABLISHED
  TCP    192.168.1.13:53081     ec2-52-54-182-58:https  ESTABLISHED
  TCP    192.168.1.13:53086     a104-86-199-105:https  ESTABLISHED
  TCP    192.168.1.13:53088     e2:https               TIME_WAIT
  TCP    192.168.1.13:53113     176.32.100.33:https    TIME_WAIT
  TCP    192.168.1.13:53119     server-54-230-87-163:https  TIME_WAIT
  TCP    192.168.1.13:53121     sfo07s17-in-f83:https  ESTABLISHED
```

Using netstat

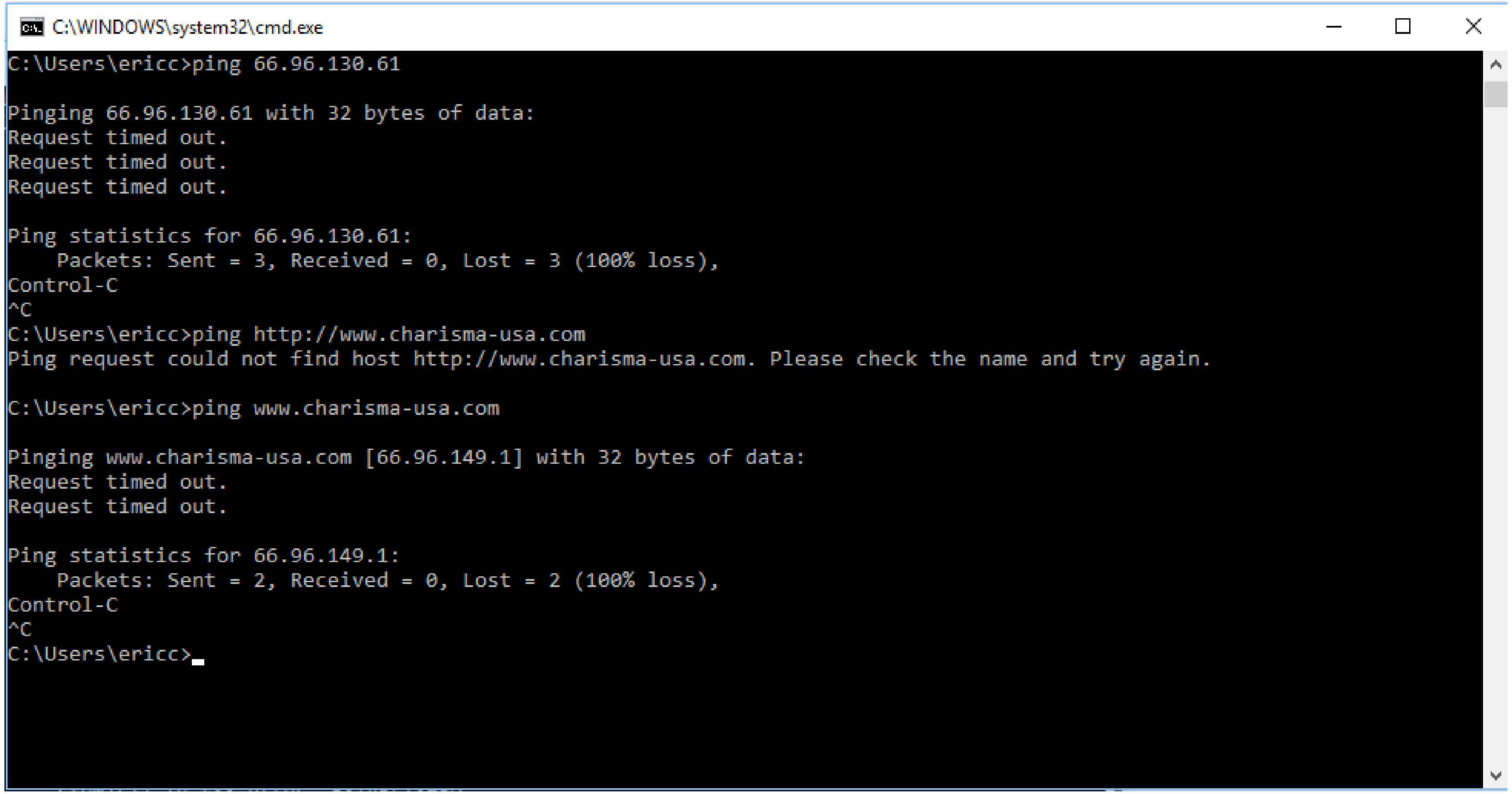
- Use 'netstat' to view active network connections
- Note: Must execute from the command shell on both Unix and Windows

Connections

- Each endpoint of a network connection is always represented by a host and **port #**
- In Python you write it out as a tuple (host,port)

```
("www.python.org",80)
("205.172.13.4",443)
```

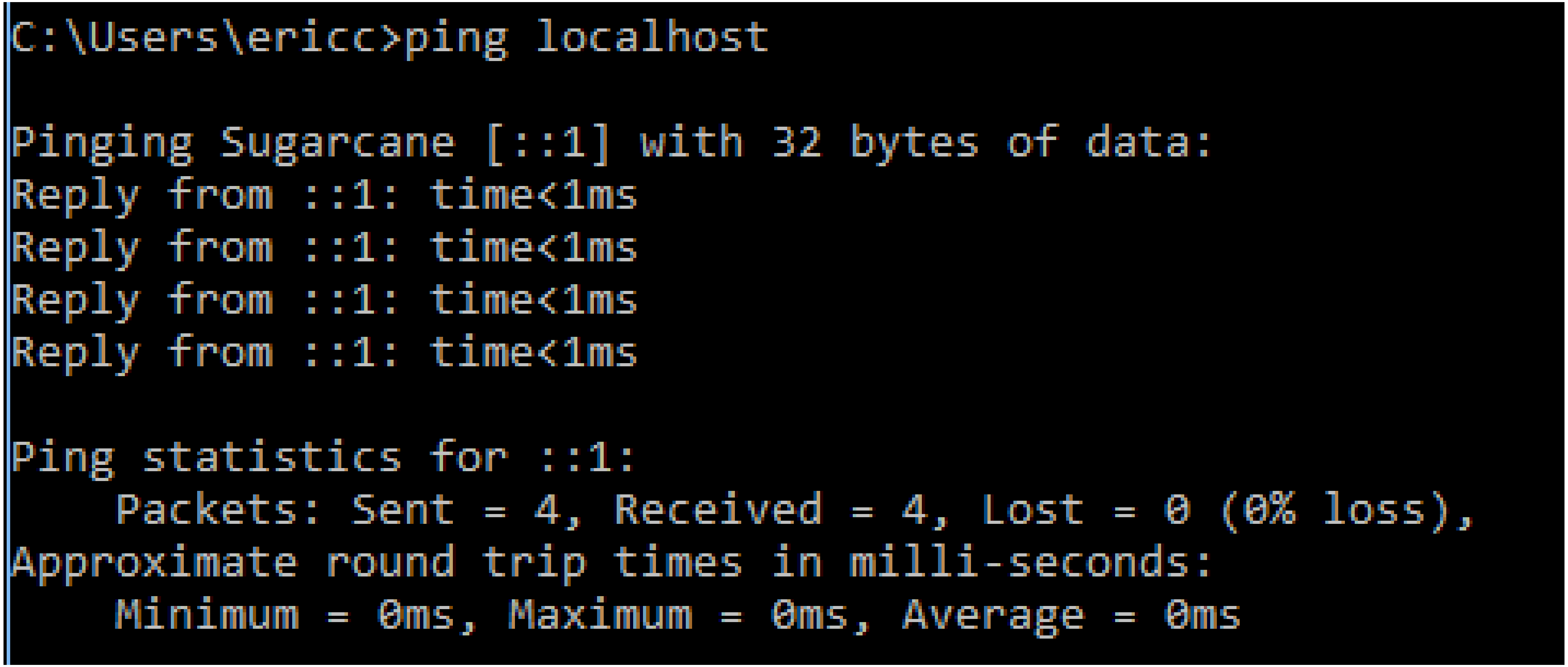
- In almost all of the network programs you’ll write, you use this convention to specify a network address



```
C:\Users\ericc>ping 66.96.130.61

Pinging 66.96.130.61 with 32 bytes of data:
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 66.96.130.61:
    Packets: Sent = 3, Received = 0, Lost = 3 (100% loss),
Control-C
^C
C:\Users\ericc>ping http://www.charisma-usa.com
Ping request could not find host http://www.charisma-usa.com. Please check the name and try again.

C:\Users\ericc>ping www.charisma-usa.com

Pinging www.charisma-usa.com [66.96.149.1] with 32 bytes of data:
Request timed out.
Request timed out.

Ping statistics for 66.96.149.1:
    Packets: Sent = 2, Received = 0, Lost = 2 (100% loss),
Control-C
^C
C:\Users\ericc>
```

Using ping

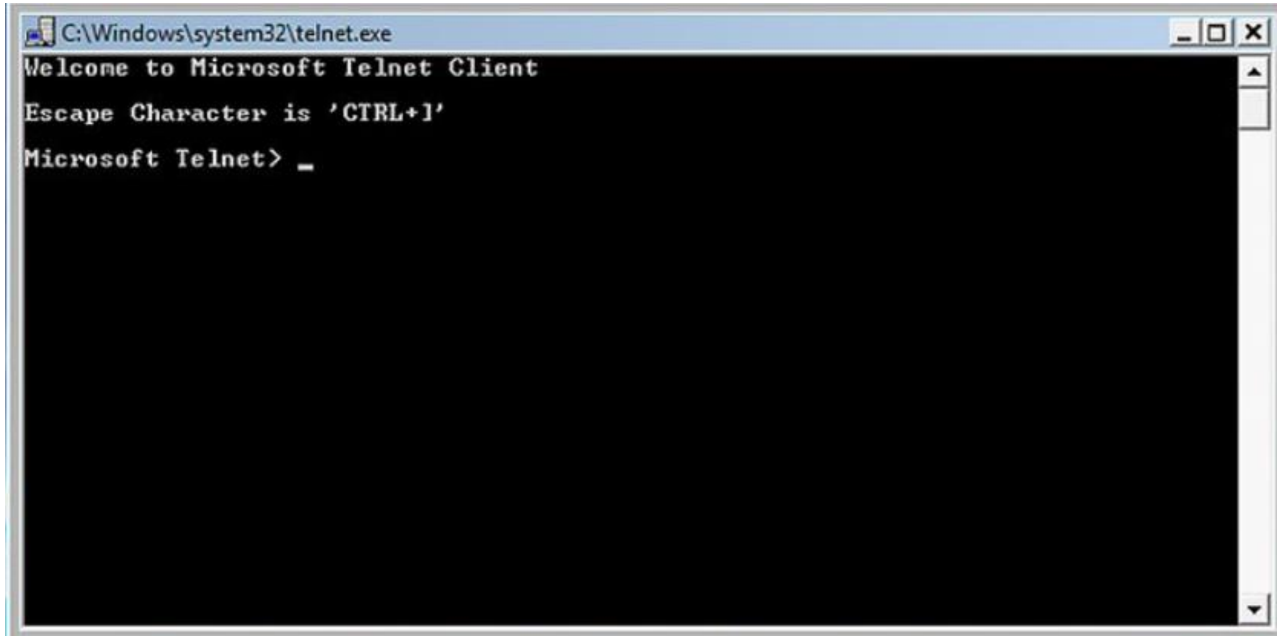
- Use ‘ping' to check if the connection to a host is love.
- Note: In the example, the connection is not built



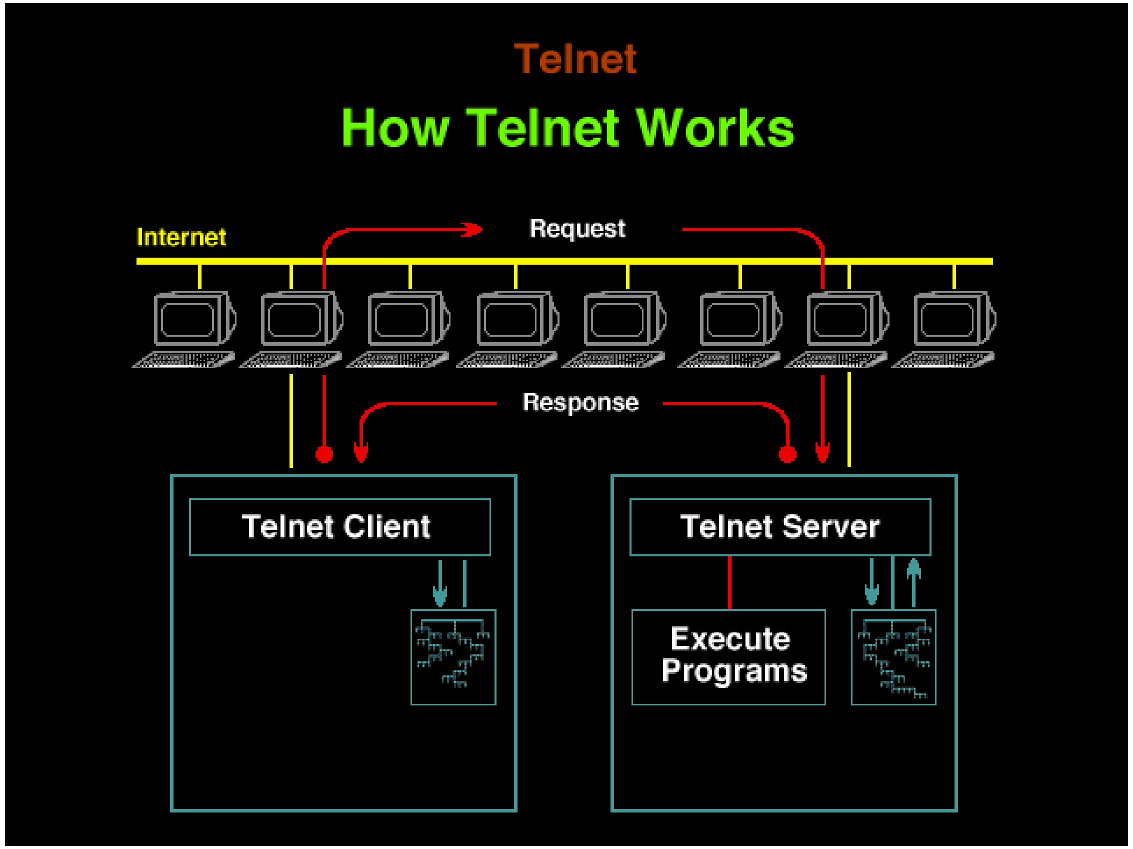
```
C:\Users\ericc>ping localhost

Pinging Sugarcane [::1] with 32 bytes of data:
Reply from ::1: time<1ms
Reply from ::1: time<1ms
Reply from ::1: time<1ms
Reply from ::1: time<1ms

Ping statistics for ::1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Even localhost fails? Why? No local host server.

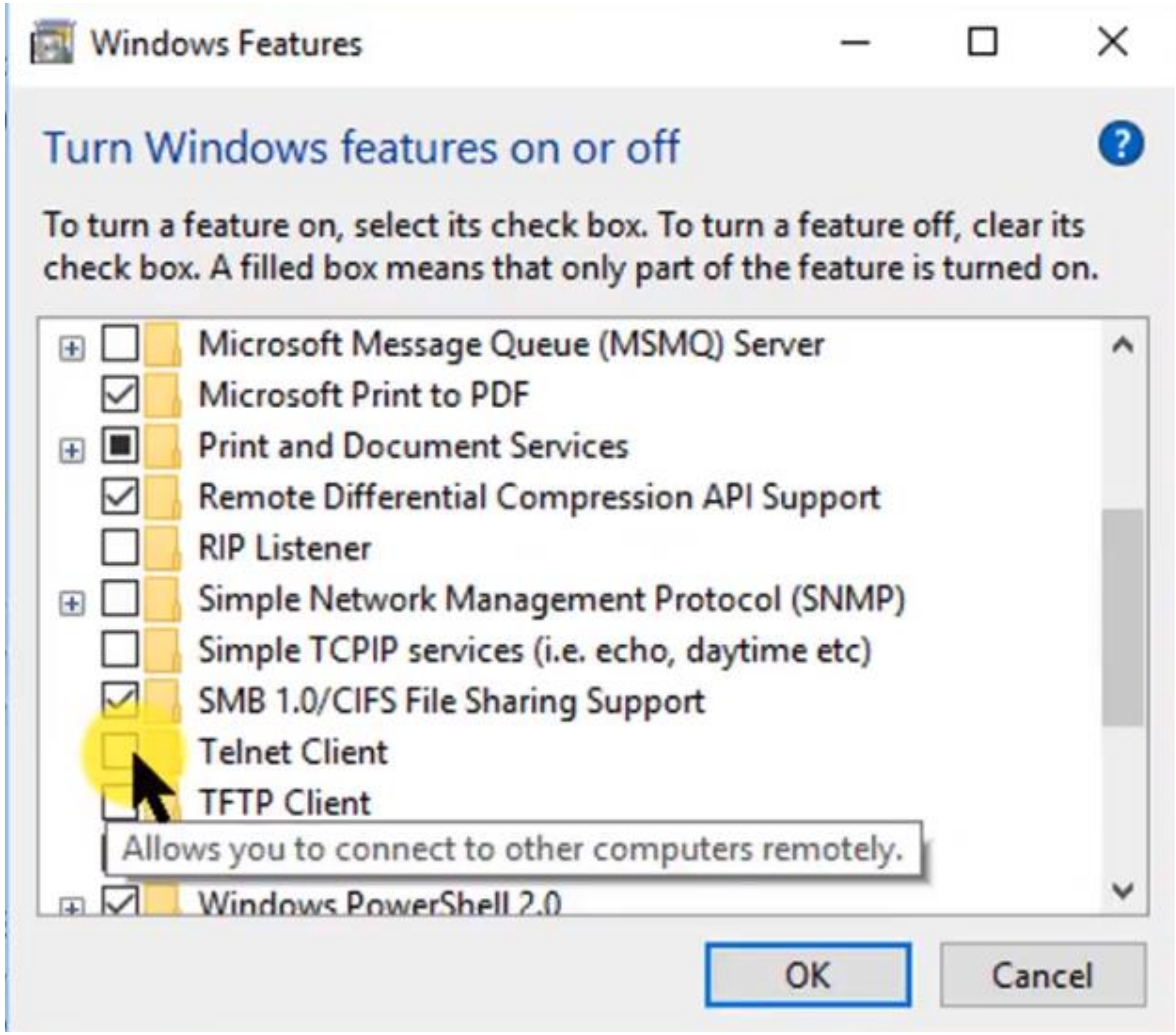
Using Telnet



```
Welcome to Microsoft Telnet Client

Escape Character is 'CTRL+]'

Microsoft Telnet> _
```



App and Features -> Program and Features -> Telnet Client



Windows Features

Turn Windows features on or off

To turn a feature on, select its check box. To turn a feature off, clear its check box. A filled box means that only part of the feature is turned on.

- ⊞ □ Microsoft Message Queue (MSMQ) Server
- ☑ Microsoft Print to PDF
- ⊞ ■ Print and Document Services
- ☑ Remote Differential Compression API Support
- □ RIP Listener
- ⊞ □ Simple Network Management Protocol (SNMP)
- □ Simple TCPIP services (i.e. echo, daytime etc)
- ☑ SMB 1.0/CIFS File Sharing Support
- □ Telnet Client
- □ TFTP Client
- ⊞ ☑ Windows PowerShell 2.0

Allows you to connect to other computers remotely.

Watch Video:
https://youtu.be/CJQfR1b43ns

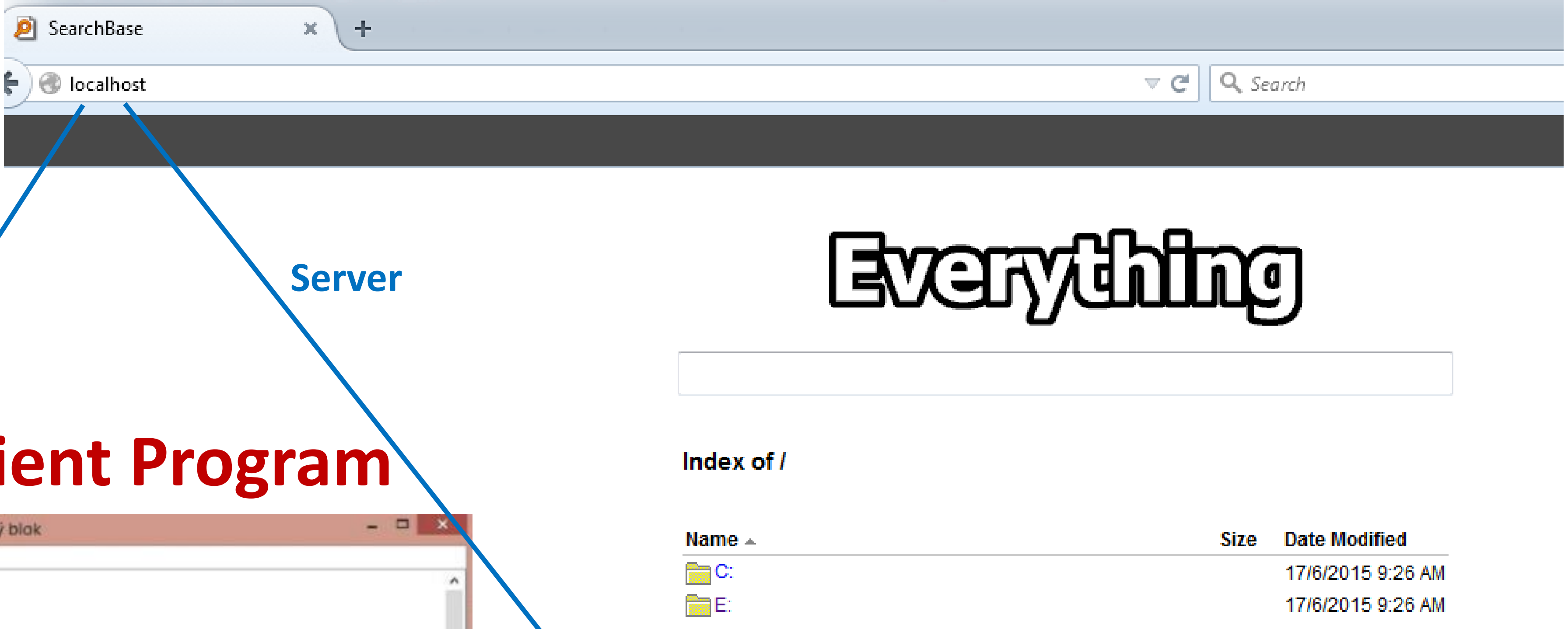
Local Host

Testing a Web Service Locally

How do you set up a local testing server?

- Local files versus remote files
- The problem with testing local files
- Running a simple local HTTP server
- Running server-side languages locally

Browser (Client)



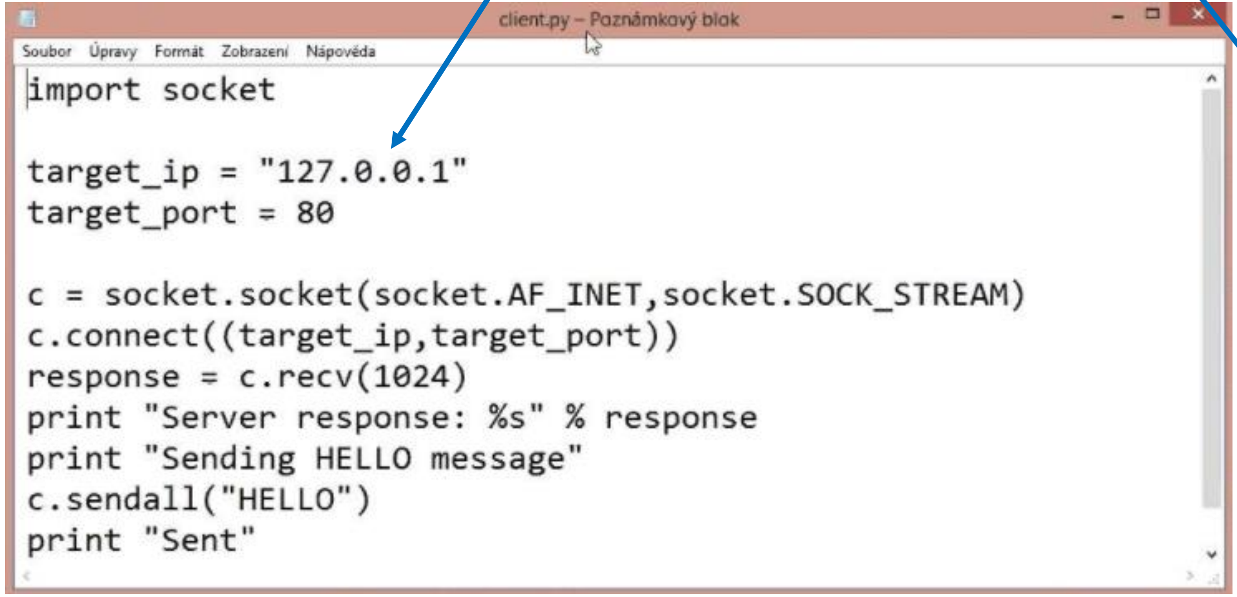
Index of /

Name ▲	Size	Date Modified
C:		17/6/2015 9:26 AM
E:		17/6/2015 9:26 AM

Client

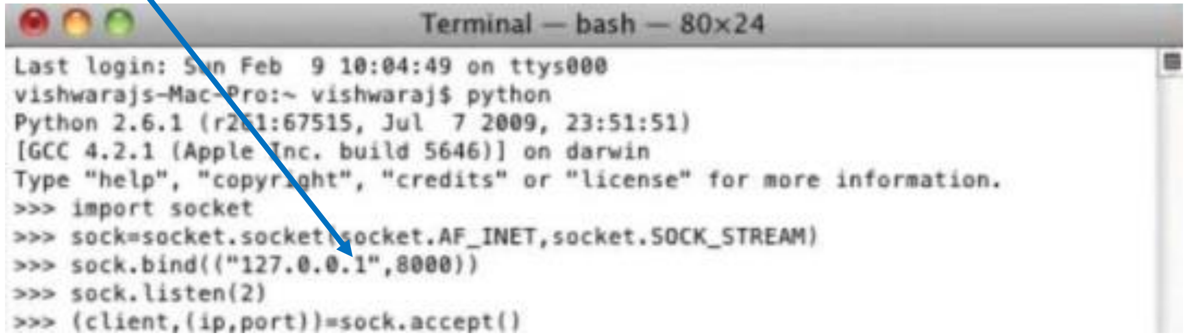
Server

Desktop Python Client Program



```
import socket

target_ip = "127.0.0.1"
target_port = 80

c = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
c.connect((target_ip,target_port))
response = c.recv(1024)
print "Server response: %s" % response
print "Sending HELLO message"
c.sendall("HELLO")
print "Sent"
```

Desktop Server Program



```
Last login: Sun Feb  9 10:04:49 on ttys000
vishwarajs-Mac-Pro:~ vishwaraj$ python
Python 2.6.1 (r261:67515, Jul  7 2009, 23:51:51)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import socket
>>> sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
>>> sock.bind(("127.0.0.1",8000))
>>> sock.listen(2)
>>> (client,(ip,port))=sock.accept()
```

Installing IIS

To install IIS:

1.In Windows, access the Control Panel and click **Add or Remove Programs**.

2.In the Add or Remove Programs window, click **Add/Remove Windows Components**.

3.Select the **Internet Information Services (IIS)** check box, click **Next**, then click **Finish**.

To learn how to use IIS, you can view the documentation at http://localhost/iishelp/iis/misc/default.asp.

Watch Video:
https://youtu.be/bJrOASXslwU

XAMPP Server Bring up Server and localhost

LECTURE 5

What is XAMPP?

- XAMPP stands for Cross-Platform (**X**), Apache (**A**), MySQL (**M**), PHP (**P**) and Perl (**P**).
- It is a simple, lightweight Apache distribution that makes it extremely easy for developers to create a local web server for testing purposes.
- Everything you need to set up a web server – server application (Apache), database (MySQL), and scripting language (PHP) – is included in a simple extractable file.
- XAMPP is also cross-platform, which means it works equally well on Linux, Mac and Windows.
- Since most actual web server deployments use the same components as XAMPP, it makes transitioning from a local test server to a live server is extremely easy as well.
- Web development using XAMPP is especially beginner friendly, as this popular PHP and MySQL for beginners course will teach you.

XAMPP

https://www.apachefriends.org/index.html

Component	On Windows	On Linux	On macOS
Apache 2.4.28	Yes	Yes	Yes
MariaDB 10.1.28	Yes	Yes	Yes
PHP	Yes - 7.1.10	Yes - 7.1.10[15]	Yes - 7.1.10[15]
phpMyAdmin	Yes - 4.7.4	Yes - 4.7.4	Yes - 4.7.4
OpenSSL	Yes - 1.0.2l	Yes - 1.0.2l	Yes - 1.0.2l
XAMPP Control Panel 3.2.2	Yes	No	No
Webalizer	Yes - 2.23-04	Yes - 2.23-05	Yes - 2.23-05
Mercury Mail	Yes	No	No
Transport System 4.63	Yes	No	No
Tomcat 7.0.56 (with mod_proxy_ajp as connector)	Yes	No	No
Strawberry Perl 7.0.56 Portable	Yes	No	No
FileZilla FTP Server 0.9.41	Yes	No	No

Installation

Watch video in the Software Installation Video Collection Course:

https://ec.teachable.com/p/software-installation-and-configuration-video-collection-free-mini-course

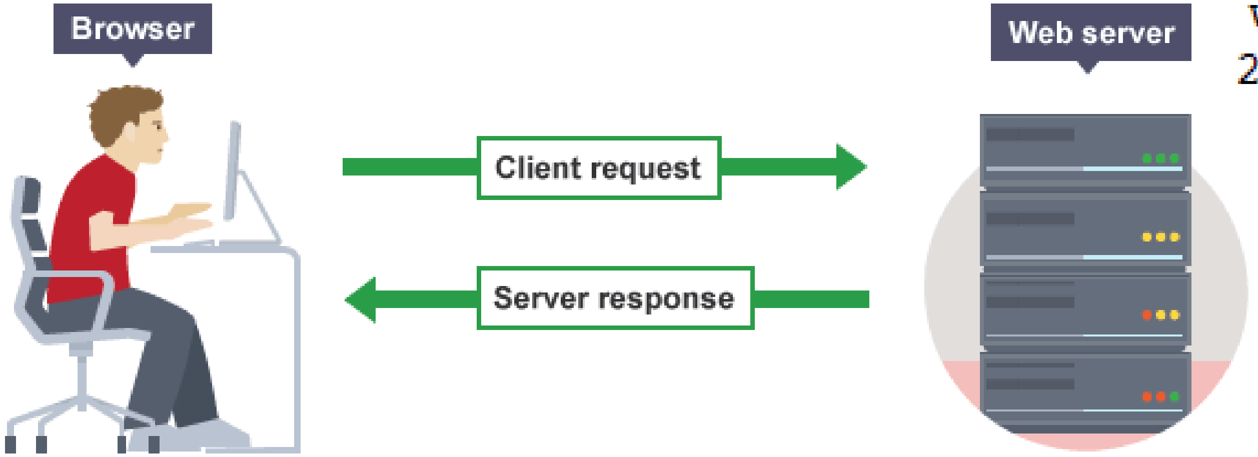
Check if the server has been brought up, especially the localhost has been brought up.

Client-Server Concept

LECTURE 6

Client/Server Concept

- Each endpoint is a running program
- Servers wait for incoming connections and provide a service (e.g., web, mail, etc.)
- Clients make connections to servers



www.bar.com
205.172.13.4
Port 80

Request/Response Cycle

- Most network programs use a request/response model based on messages
- Client sends a request message (e.g., HTTP)

```
GET /index.html HTTP/1.0
```

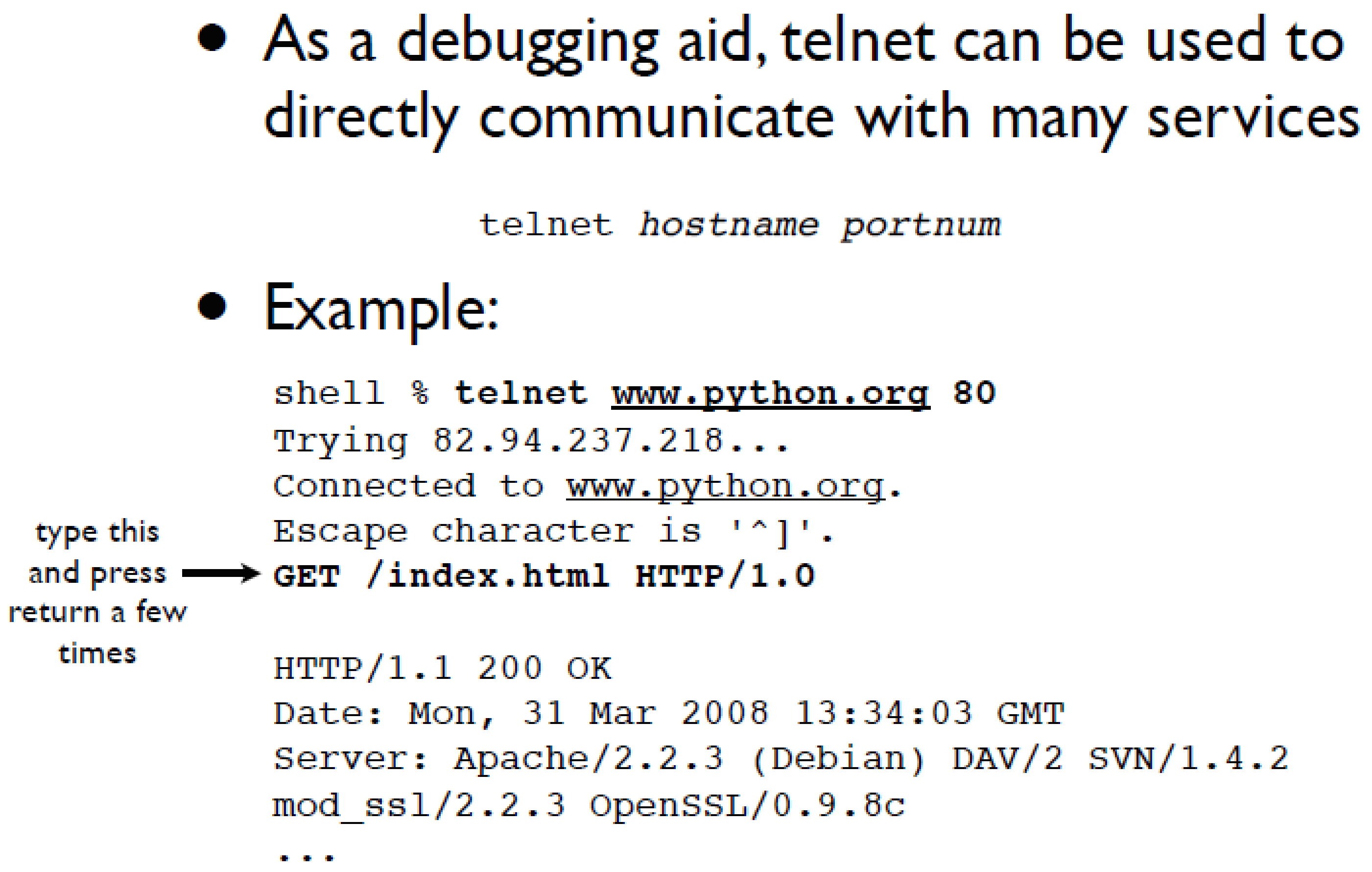
- Server sends back a response message

```
HTTP/1.0 200 OK
Content-type: text/html
Content-length: 48823
<HTML>
...
```

- The exact format depends on the application

Using Telnet

on Linux/Unix



- As a debugging aid, telnet can be used to directly communicate with many services

```
telnet hostname portnum
```

- Example:

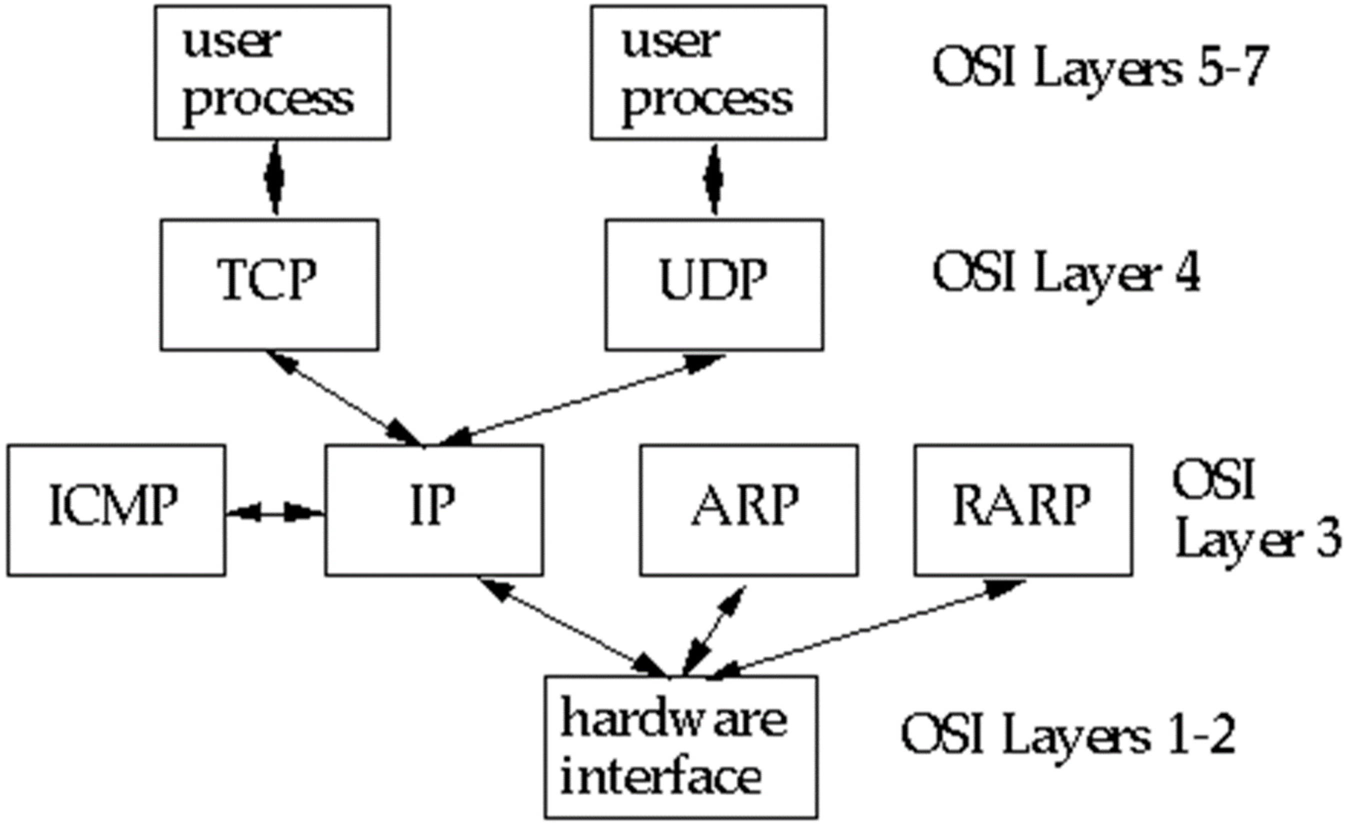
type this and press return a few times →

```
shell % telnet www.python.org 80
Trying 82.94.237.218...
Connected to www.python.org.
Escape character is '^]'.
GET /index.html HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 31 Mar 2008 13:34:03 GMT
Server: Apache/2.2.3 (Debian) DAV/2 SVN/1.4.2
mod_ssl/2.2.3 OpenSSL/0.9.8c
...
```

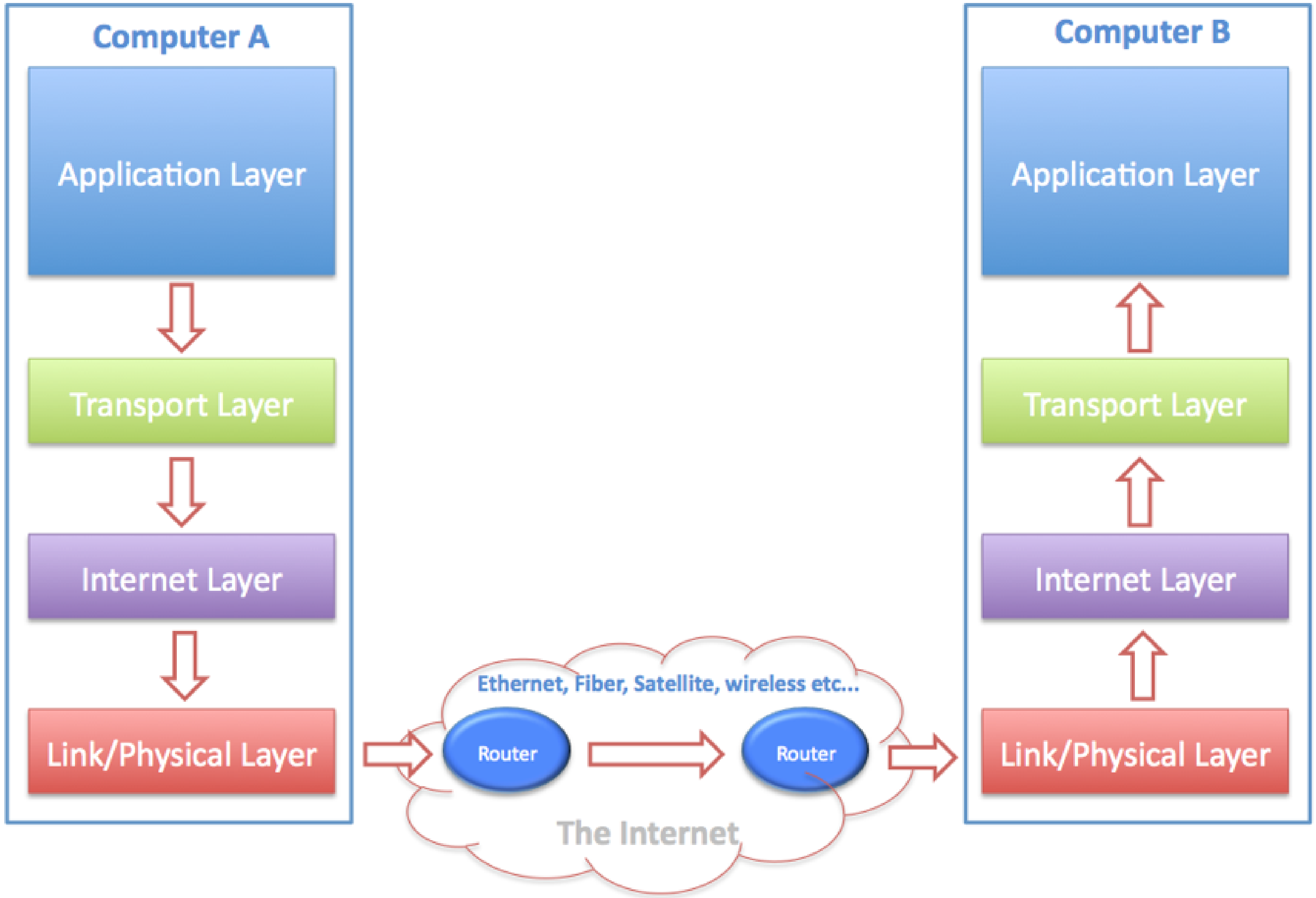
Data Transport

- There are two basic types of communication
- **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.

OSI model		TCP / IP model
Application	7.	Application
Presentation	6.	
Session	5.	
Transport	4.	Transport
Network	3.	Internet
Data link	2.	Host-to-Network
Physical	1.	



Data Transmission over the Internet through TCP/IP



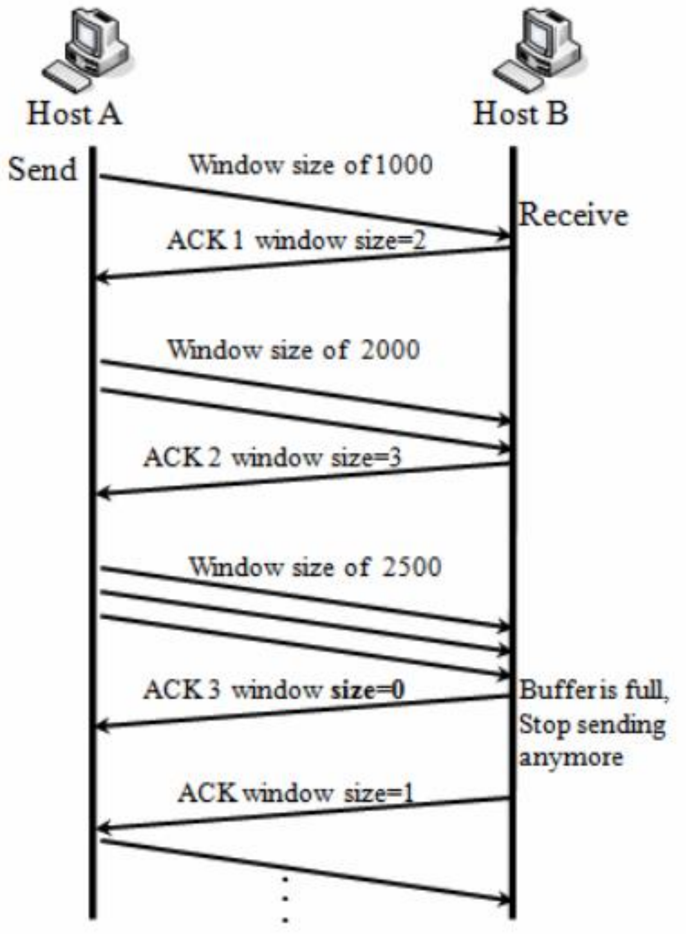
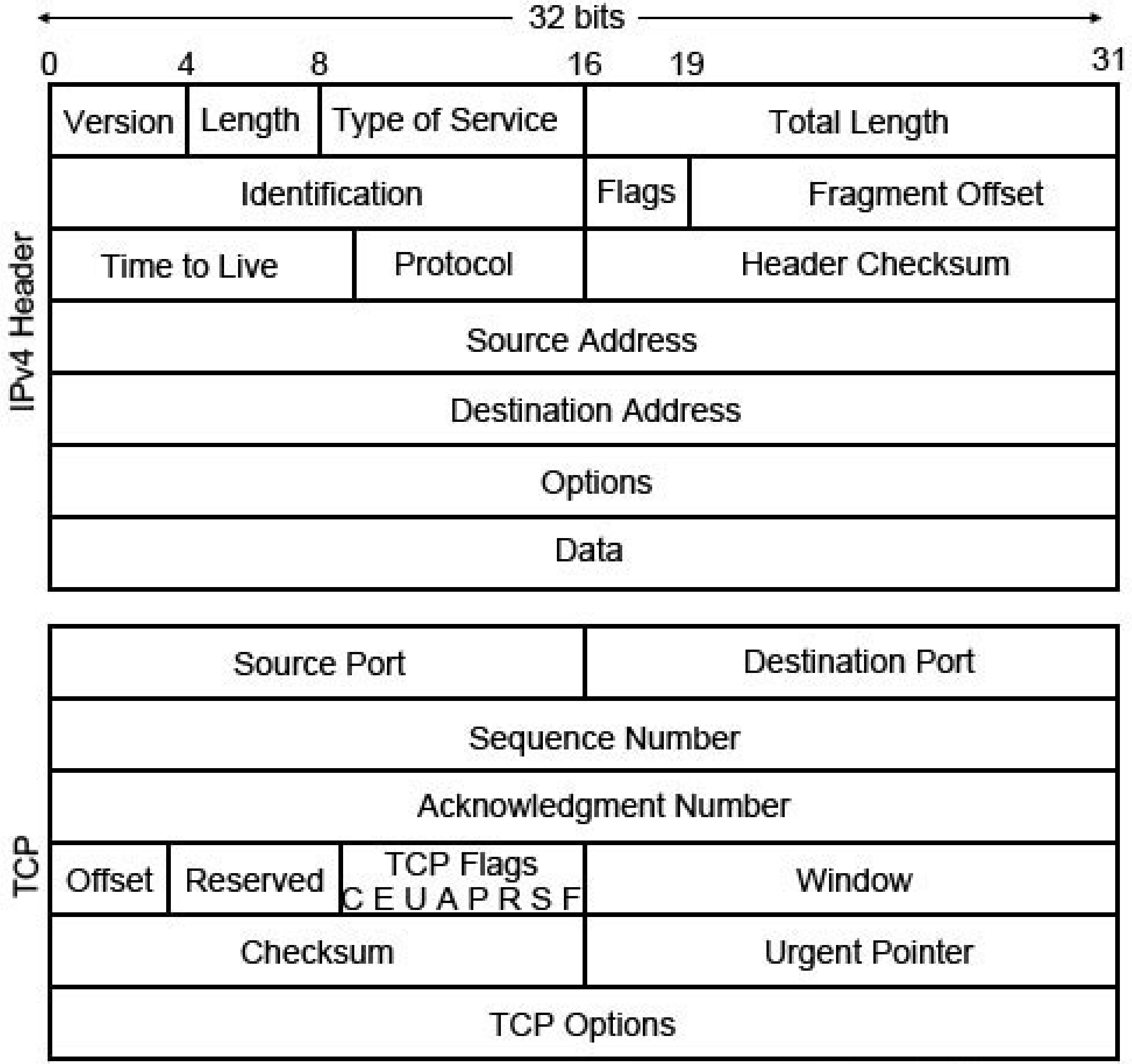


Figure 2.2. TCP flow control using *windowing*

TCP/IP Packet



32 bits

0 4 8 16 19 31

IPv4 Header

Version | Length | Type of Service | Total Length

Identification | Flags | Fragment Offset

Time to Live | Protocol | Header Checksum

Source Address

Destination Address

Options

Data

TCP

Source Port | Destination Port

Sequence Number

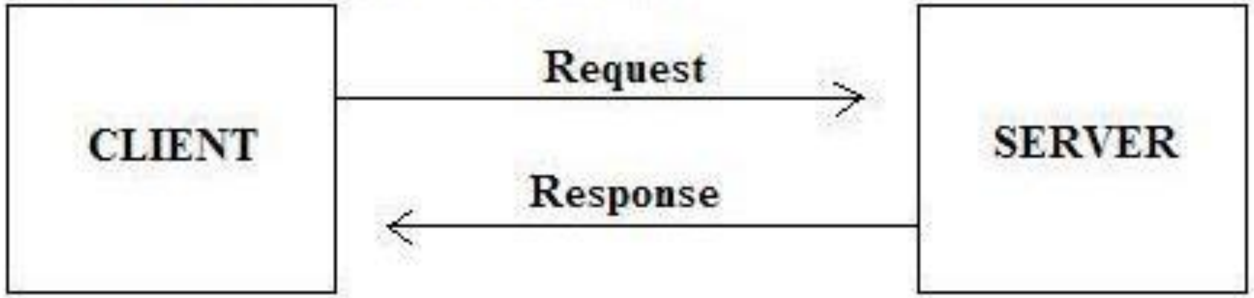
Acknowledgment Number

Offset | Reserved | TCP Flags C E U A P R S F | Window

Checksum | Urgent Pointer

TCP Options

UDP Request / Response Paradigm



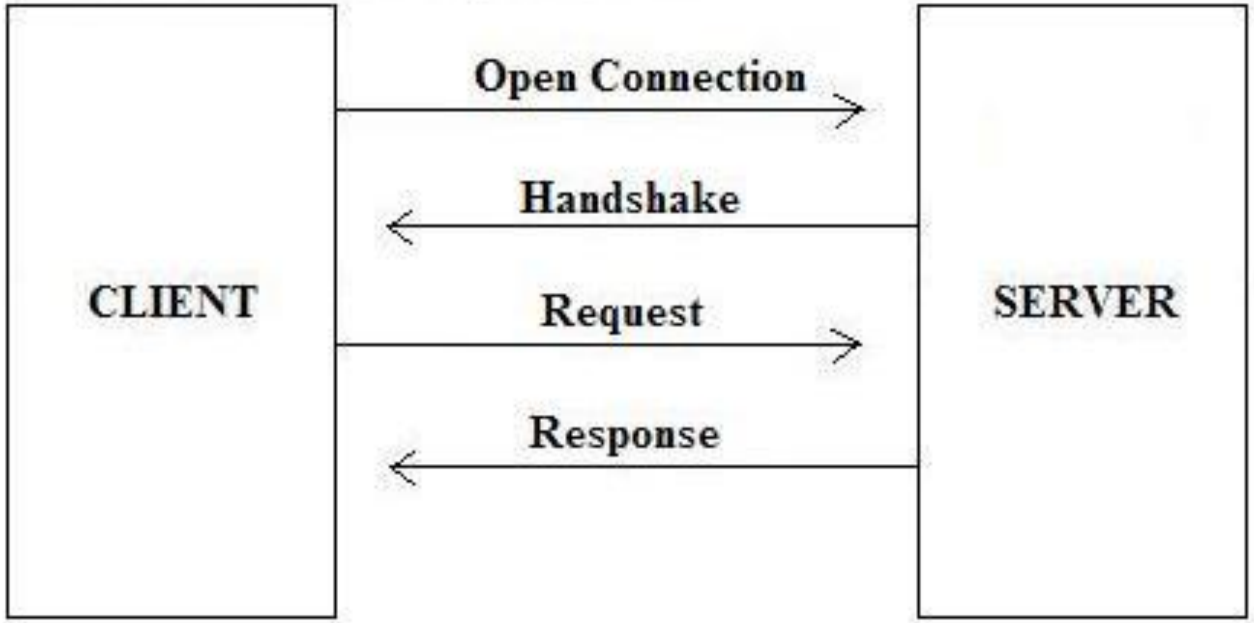
CLIENT

Request

Response

SERVER

TCP Handshake Paradigm



CLIENT

Open Connection

Handshake

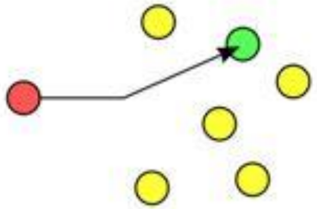
Request

Response

SERVER

TCP

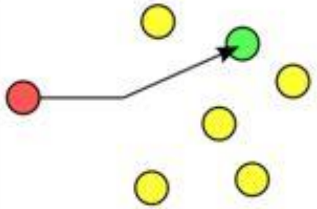
- **Slower but reliable transfers**
- **Typical applications:**
  - Email
  - Web browsing



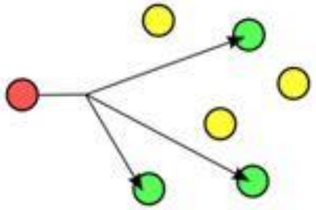
unicast

UDP

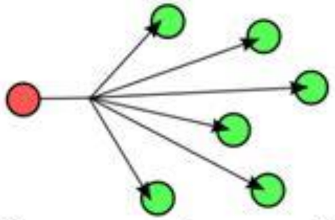
- **Fast but non-guaranteed transfers (“best effort”)**
- **Typical applications:**
  - VoIP
  - Music streaming



unicast



multicast



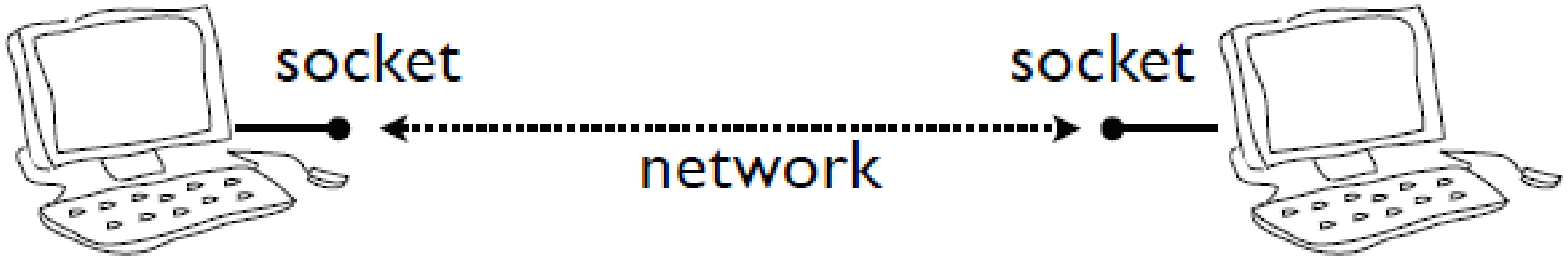
broadcast

Socket (Client)

LECTURE 7

Sockets

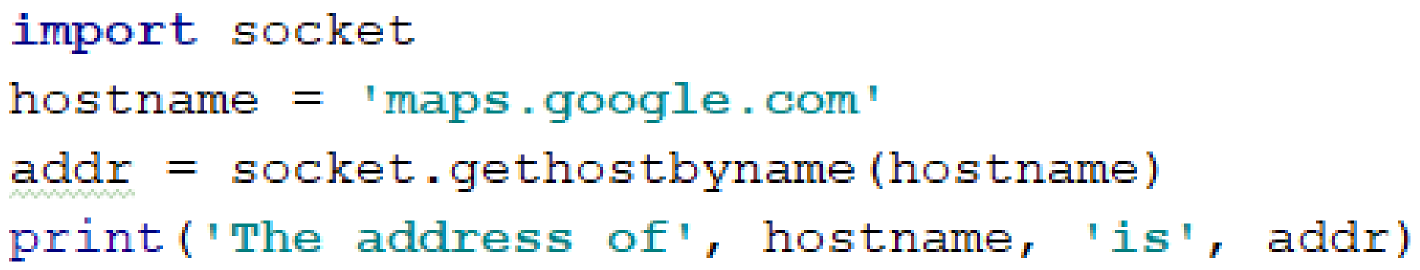
- Programming abstraction for network code
- Socket: A communication endpoint



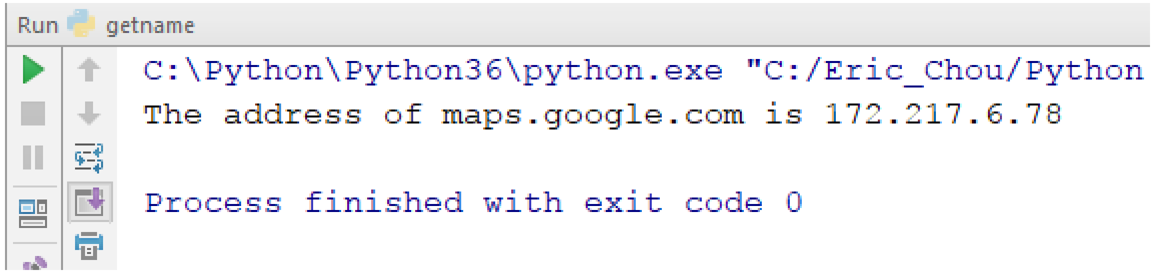
- Supported by **socket** library module
- Allows connections to be made and data to be transmitted in either direction

Get the Host Name

Demo Program: getname.py



```
import socket
hostname = 'maps.google.com'
addr = socket.gethostbyname(hostname)
print('The address of', hostname, 'is', addr)
```



```
C:\Python\Python36\python.exe "C:/Eric_Chou/Python
The address of maps.google.com is 172.217.6.78

Process finished with exit code 0
```

Socket Basics

- To create a socket

```
import socket
s = socket.socket(addr_family, type)
```

- Address Familier

socket.AF_INET	Internet protocol (IPv4)
socket.AF_INET6	Internet protocol (IPv6)

- Socket types

socket.SOCK_STREAM	Connection based stream (TCP)
socket.SOCK_DGRAM	Datagrams (UDP)

- Example:

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
```

Socket Types

- Most common case: TCP connection

```
from socket import *
s = socket(AF_INET, SOCK_STREAM)      # TCP
s = socket(AF_INET, SOCK_DGRAM)       # UDP
```

- Almost all code will use one of following

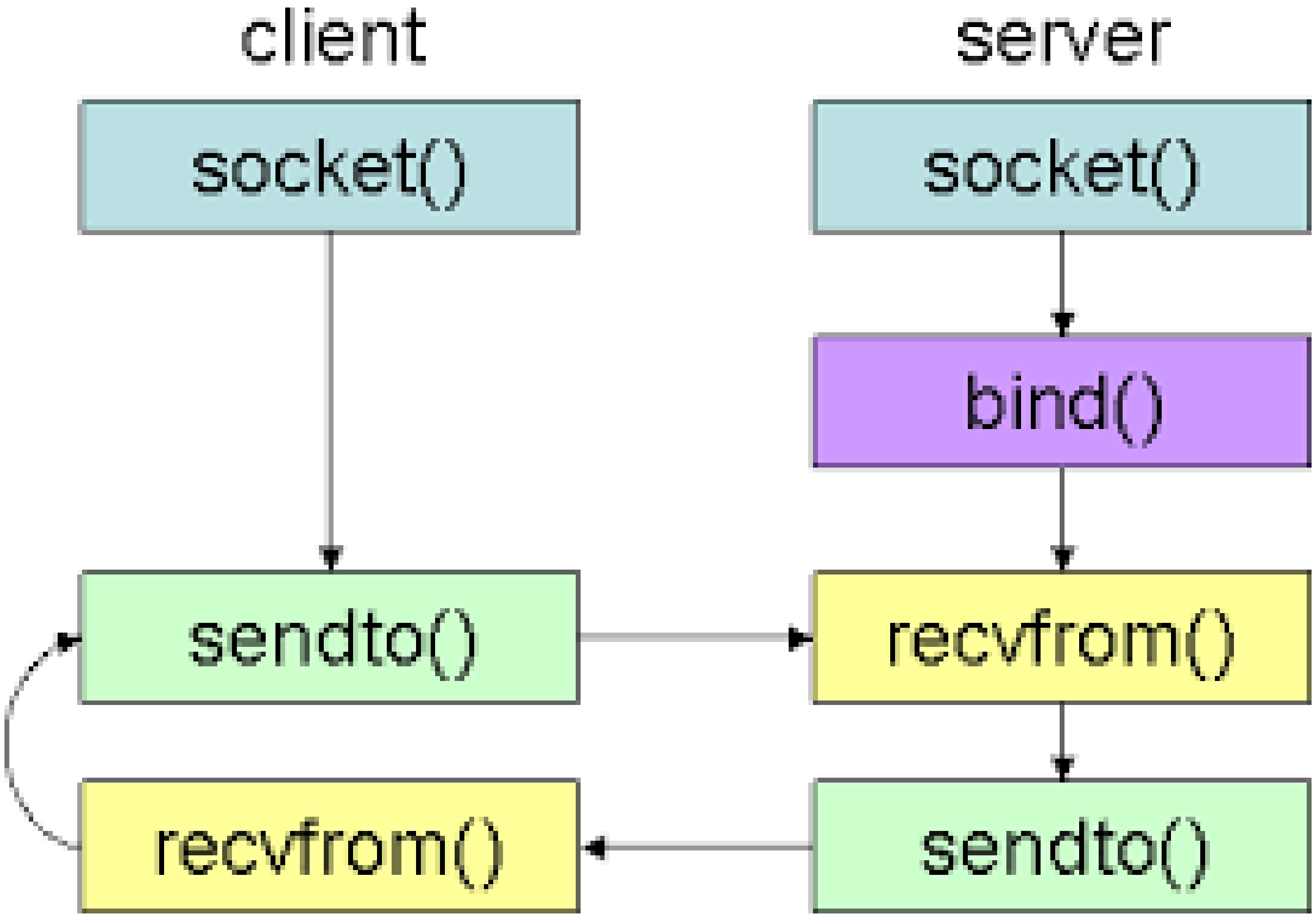
```
s = socket(AF_INET, SOCK_STREAM)      # TCP
```

Using a Socket

- Creating a socket is only the first step

```
s = socket(AF_INET, SOCK_STREAM)
```

- Further use depends on application
- Server
  - Listen for incoming connections
- Client
  - Make an outgoing connection


client
server
socket()
socket()
bind()
sendto()
recvfrom()
recvfrom()
sendto()

TCP Client

- How to make an outgoing connection

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.connect(("www.python.org",80)) # Connect
s.send(bytes("GET /index.html HTTP/1.0\n\n“), ‘utf8’) # Send request
data = s.recv(10000) # Get response
s.close()
```

- **s.connect(addr)** makes a connection

```
s.connect(("www.python.org",80))
```

- Once connected, use **sendto()**, **recvfrom()** to transmit and receive data
- **close()** shuts down the connection

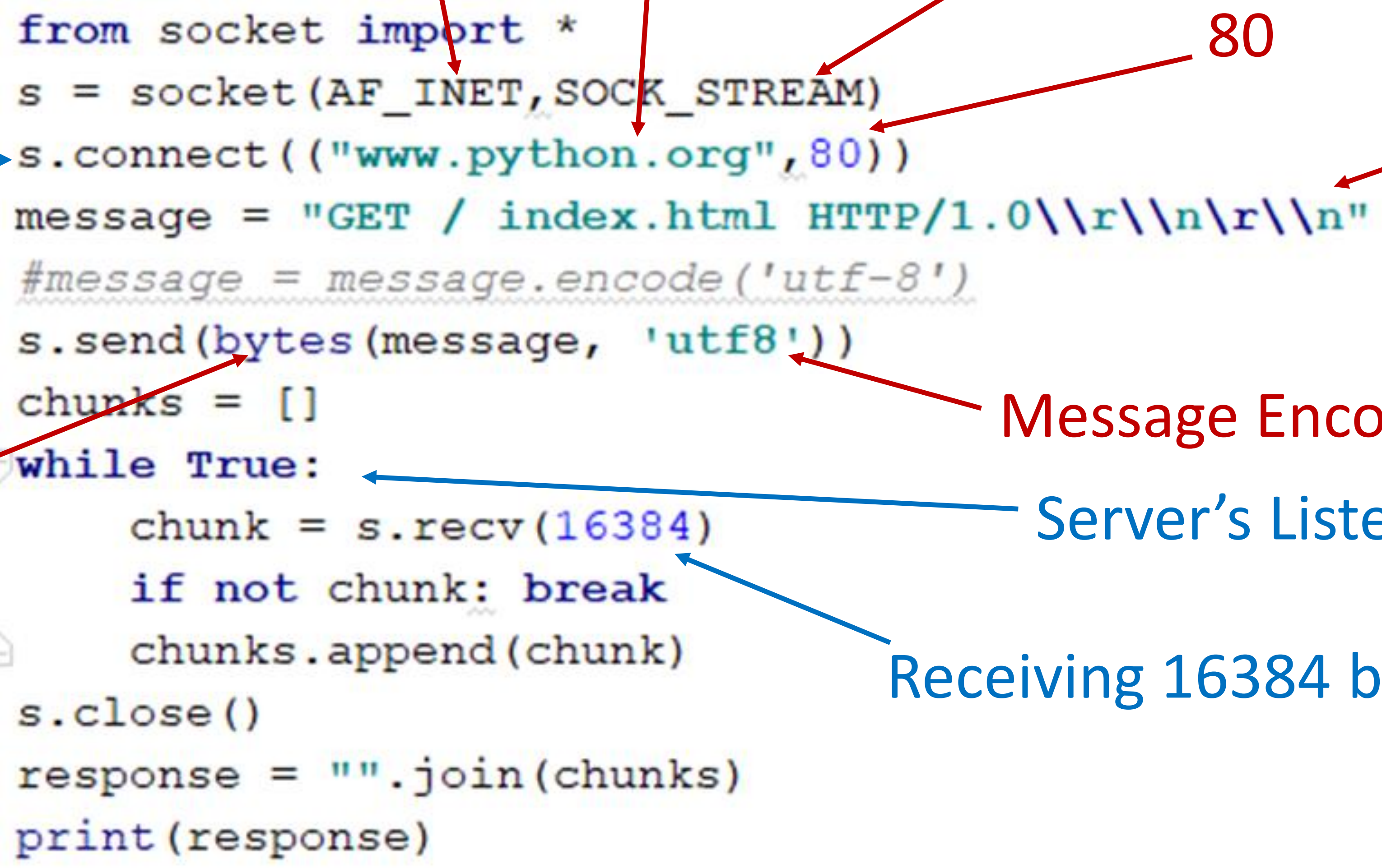
Basic Example (Client Side)

Demo Program: basic0.py (can’t run by itself)

Objectives:

- Low-level network programming with sockets
- How to connect to a TCP server

- This code is fairly typical for TCP client code.
- Once connected to a server, use **send**() to send request data. To read a response, you will typically have to read data in chunks with multiple **recv**() operations.
- **recv**() returns an empty string to signal the end of data (i.e., if the server closed its end of the connection).
- Recall that using the string **join**() method is significantly faster than using string concatenation (+) to join string fragments together.



```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.connect(("www.python.org",80))
message = "GET / index.html HTTP/1.0\\r\\n\r\\n"
#message = message.encode('utf-8')
s.send(bytes(message, 'utf8'))
chunks = []
while True:
    chunk = s.recv(16384)
    if not chunk: break
    chunks.append(chunk)
s.close()
response = "".join(chunks)
print(response)
```

Host domain name

IPv4

Socket Declaration

TCP

80

Socket Connection

Message

Sending

Message Encoding

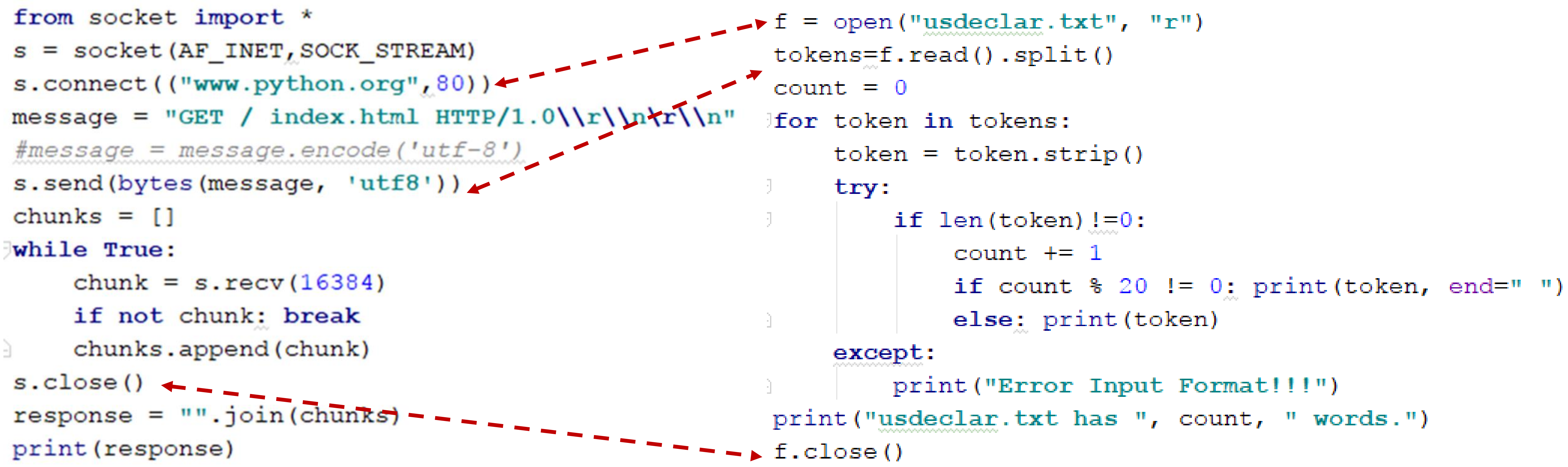
Server’s Listening Loop

Byte by byte

Receiving 16384 bytes

Socket Closed

Analogy between File I/O Stream and Socket I/O Stream



```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.connect(("www.python.org",80))
message = "GET / index.html HTTP/1.0\\r\\n\\r\\n"
#message = message.encode('utf-8')
s.send(bytes(message, 'utf8'))
chunks = []
while True:
    chunk = s.recv(16384)
    if not chunk: break
    chunks.append(chunk)
s.close()
response = "".join(chunks)
print(response)
```

```
f = open("usdeclar.txt", "r")
tokens=f.read().split()
count = 0
for token in tokens:
    token = token.strip()
    try:
        if len(token)!=0:
            count += 1
            if count % 20 != 0: print(token, end=" ")
            else: print(token)
    except:
        print("Error Input Format!!!")
print("usdeclar.txt has ", count, " words.")
f.close()
```

Socket (Server)

LECTURE 8

Server Implementation

- Network servers are a bit more tricky
- Must listen for incoming connections on a well-known port number
- Typically run forever in a server-loop
- May have to service multiple clients

TCP Server

- A simple server

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print("Received connection from", a)
    c.send("Hello %s\n" % a[0])
    c.close()
```

maximum number of queued connections and should be at least 1

- Send a message back to a client

```
% telnet localhost 9000
Connected to localhost.
Escape character is '^]'.
Hello 127.0.0.1
Connection closed by foreign host.
```

Server Message

TCP Server

- Address binding

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c, a = s.accept()
    print("Received connection from",  a)
    c.send("Hello %s\n" % a[0])
    c.close()
```

binds the socket to a specific address

- Addressing

```
s.bind(("",9000))
s.bind(("localhost",9000))
s.bind(("192.168.2.1",9000))
s.bind(("104.21.4.2",9000))
```

binds to local host

If system has multiple IP addresses, can bind to a specific address

TCP Server

- Start listening for connections

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print("Received connection from", a)
    c.send("Hello %s\n" % a[0])
    c.close()
```

- s.listen(backlog)
- backlog is # of pending connections to allow
- Note: not related to max number of clients

Tells operating system to start listening for connections on the socket

TCP Server

- Accepting a new connection

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c, a = s.accept()
    print("Received connection from", a)
    c.send("Hello %s\n" % a[0])
    c.close()
```

- s.accept() blocks until connection received
- Server sleeps if nothing is happening

Accept a new client connection

TCP Server

- Client Socket and address pair

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c, a = s.accept()
    print("Received connection from", a)
    c.send("Hello %s\n" % a[0])
    c.close()
```

Accept returns a pair (client_socket, addr)

<socket._socketobject ("104.23.11.4",27743) object at 0x3be30>
This is a new socket that's used for data

("104.23.11.4",27743)
This is the network/port address of the client that connected

TCP Server

- Client Socket and address pair

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c, a = s.accept()
    print("Received connection from", a)
    c.send("Hello %s\n" % a[0])
    c.close()
```

Note: Use the client socket for transmitting data. The server socket is only used for accepting new connections.

Send data to client

TCP Server

- Client Socket and address pair

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c, a = s.accept()
    print("Received connection from", a)
    c.send("Hello %s\n" % a[0])
    c.close()
```

Close client connection

- Note: Server can keep client connection alive as long as it wants
- Can repeatedly receive/send data

TCP Server

- Client Socket and address pair

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c, a = s.accept()
    print("Received connection from", a)
    c.send("Hello %s\n" % a[0])
    c.close()
```

Wait for next connection

- Original server socket is reused to listen for more connections
- Server runs forever in a loop like this

Simple Client-Server Sockets Example

LECTURE 9

Simple Client Server Programs

Demo Program: basic1s0.py (Server Program), basic1c0.py (client program) Watch Video: client_server.wmv

Server (PyCharm)	Client (IDLE)

from socket import *
s = socket(AF_INET, SOCK_STREAM)
s.bind(("",15000))
s.listen(5)
c, a = s.accept()
	from socket import *
	s = socket(AF_INET, SOCK_STREAM)
	s.connect(("localhost",15000))

c	
a	
	s.send(bytes("Hello World“, ‘utf8’))
data = c.recv(1024)	
data	

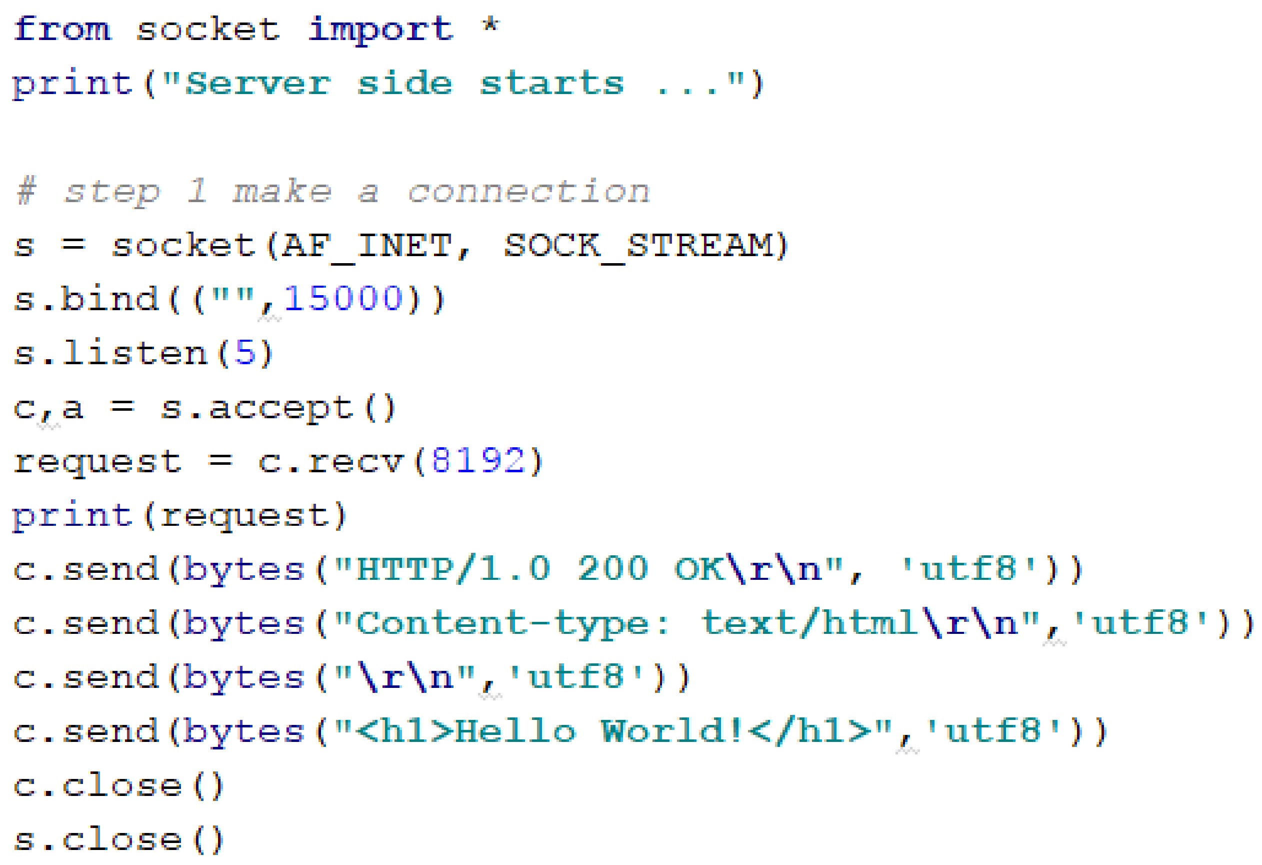
Simple Client Server Programs

Demo Program: basic1.py (client program)

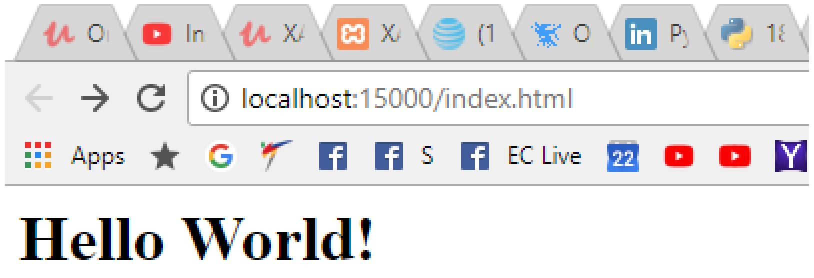
Server (PyCharm)	Client (IDLE)
c.send(bytes("Hello Yourself“, ‘utf8’))	
	resp = s.recv(1024)
	Resp
	s.recv(1024)
c.send(bytes("Goodbye“, ‘utf8’))	
c.close()	
	s.recv(1024)
	s.recv(1024)

Send a Web-page to a Web-site Connecting to server

Demo Program: browse.py



```
from socket import *
print("Server side starts ...")

# step 1 make a connection
s = socket(AF_INET, SOCK_STREAM)
s.bind(("",15000))
s.listen(5)
c,a = s.accept()
request = c.recv(8192)
print(request)
c.send(bytes("HTTP/1.0 200 OK\r\n", 'utf8'))
c.send(bytes("Content-type: text/html\r\n",'utf8'))
c.send(bytes("\r\n",'utf8'))
c.send(bytes("<h1>Hello World!</h1>",'utf8'))
c.close()
s.close()
```



localhost:15000/index.html

Hello World!

Connecting to the Website

Server (PyCharm)	Client (Chrome Browser)

from socket import *
s = socket(AF_INET, SOCK_STREAM)
s.bind(("",15000))
s.listen(5)
c, a = s.accept()	

	http://localhost:15000/index.html
request = c.recv(8192)	
print(request)	

c.send(bytes("HTTP/1.0 200 OK\r\n", ‘utf8’))
c.send(bytes("Content-type: text/html\r\n", ‘utf8’))
c.send(bytes("\r\n", ‘utf8’))
c.send(bytes("<h1>Hello World</h1>", ‘utf8’))
c.close()	