COMP2602 Computer Networks

Chapter 2
Application Layer

A note on the use of these ppt slides:

We're making these slides freely available to all (faculty, students, readers). They're in powerpoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- ❑ If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
- ❑If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

*Computer Networking:
A Top Down Approach
Featuring the Internet,*
Jim Kurose, Keith Ross
Addison-Wesley, 2022

Chapter 2: Application Layer

Our goals:

- conceptual, implementation aspects of network application protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- learn about protocols by examining popular application-level protocols
  - HTTP-Hypertext Transfer Protocol
  - FTP
  - SMTP / POP3 / IMAP
  - DNS
- programming network applications
  - socket API

SMTP-Simple Mail Transfer Protocol, POP3-Post Office Protocol, IMAP-Internet Message Access Protocol

Network applications: some jargon

Process: program running within a host.

- within same host, two processes communicate using interprocess communication (defined by OS).
- processes running in different hosts communicate with an application-layer protocol

user agent: interfaces with user “above” and network “below”.

- implements user interface & application-level protocol
  - Web: browser
  - E-mail: mail reader
  - streaming audio/video: media player

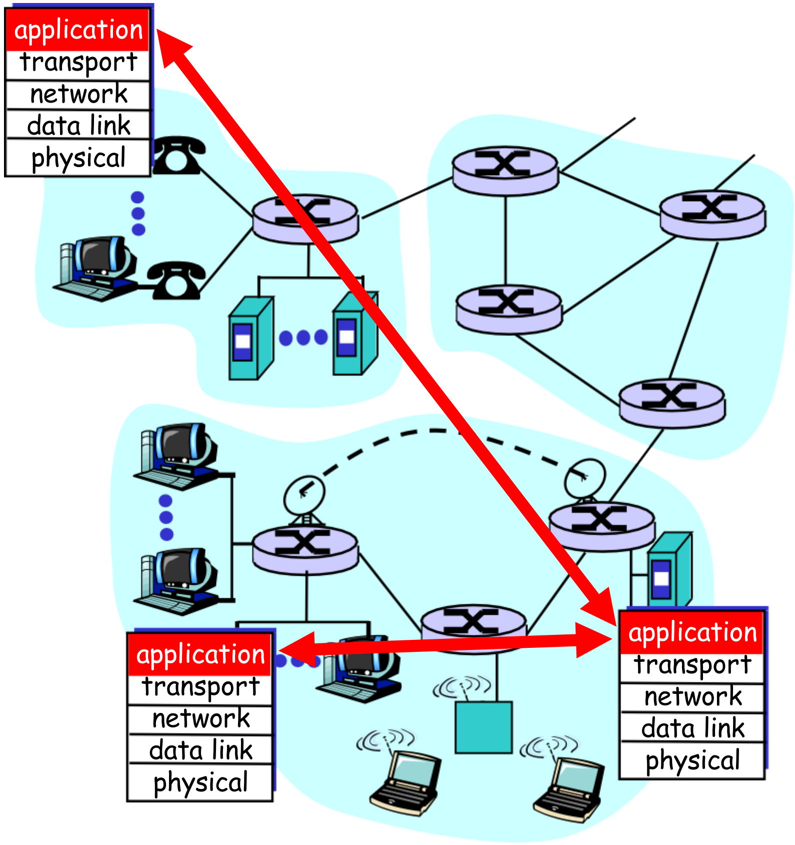
Applications and application-layer protocols

Application: communicating, distributed processes

- e.g., e-mail, Web, P2P file sharing, instant messaging
- running in end systems (hosts)
- exchange messages to implement application

Application-layer protocols

- one "piece" of an app
- define messages exchanged by apps and actions taken
- use communication services provided by lower layer protocols (TCP, UDP)



App-layer protocol defines

- Types of messages exchanged, eg, request & response messages
- Syntax of message types: what fields in messages & how fields are delineated
- Semantics of the fields, ie, meaning of information in fields
- Rules for when and how processes send & respond to messages

Public-domain protocols:

- defined in RFCs
- allows for interoperability
- eg, HTTP, SMTP

Proprietary protocols:

- eg, BitTorrent

Client-server paradigm

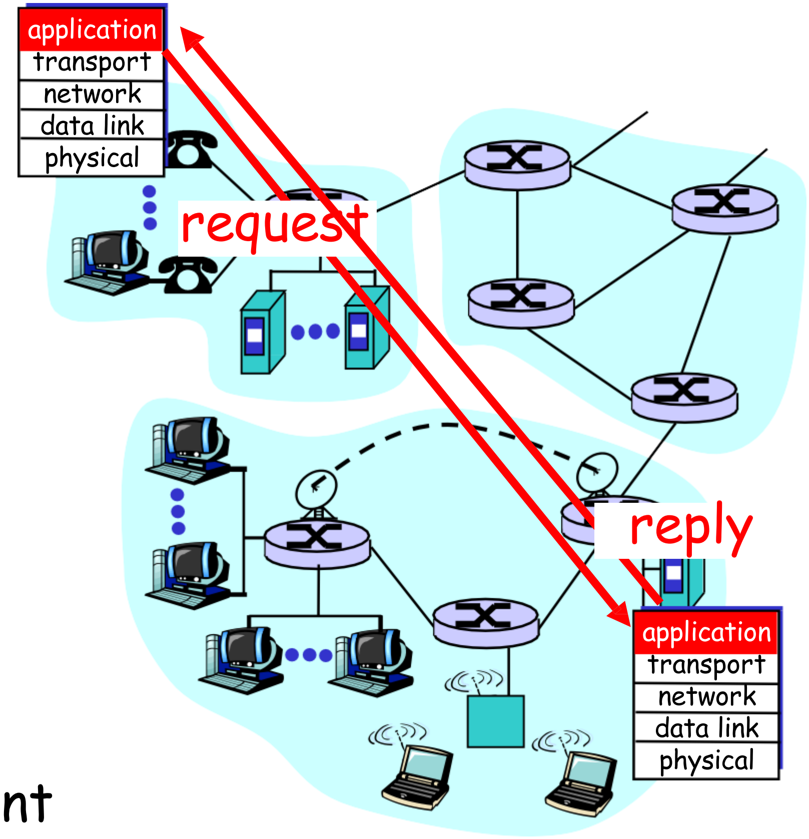
Typical network app has two pieces: *client* and *server*

Client:

- ❐ initiates contact with server ("speaks first")
- ❐ typically requests service from server,
- ❐ Web: client implemented in browser; e-mail: in mail reader

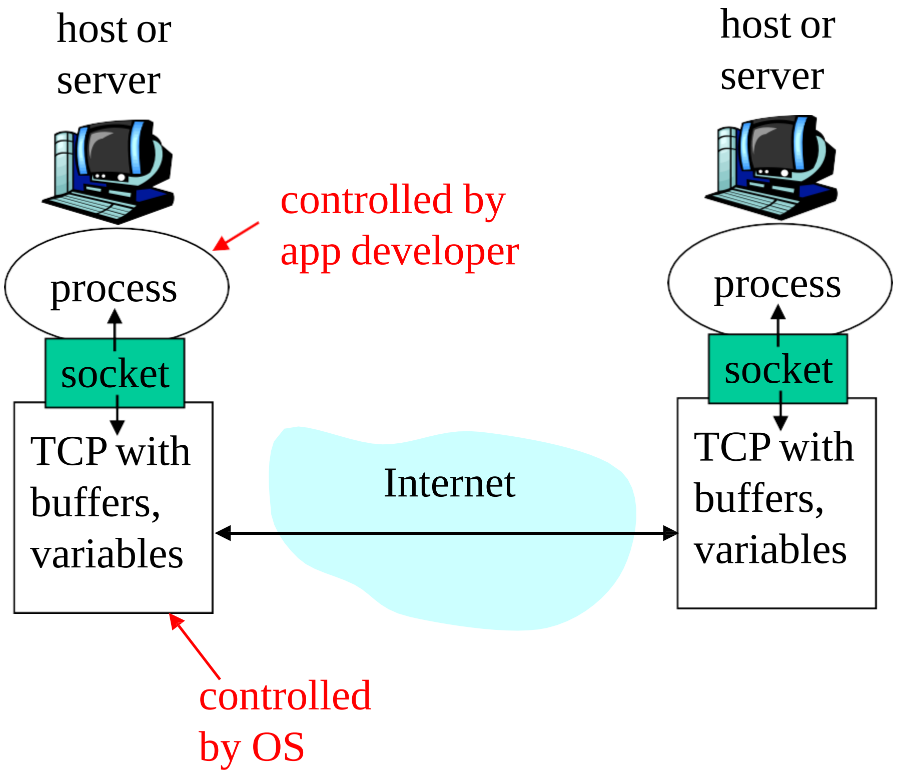
Server:

- ❐ provides requested service to client
- ❐ e.g., Web server sends requested Web page, mail server delivers e-mail



Processes communicating across network

- process sends/receives messages to/from its socket
- 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 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

- API: (1) choice of transport protocol; (2) ability to fix a few parameters (lots more on this later)

Addressing processes:

- For a process to receive messages, it must have an identifier
- Every host has a unique 32-bit IP address
- Q: does the IP address of the host on which the process runs suffice for identifying the process?
- Answer: No, many processes can be running on same host
- Identifier includes both the IP address and port numbers associated with the process on the host.
- Example port numbers:
  - HTTP server: 80
  - Mail server: 25
- More on this later

What transport service does an app need?

Data loss

- some apps (e.g., audio) can tolerate some loss
- other apps (e.g., file transfer, telnet) require 100% reliable data transfer

Timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"

Bandwidth

- some apps (e.g., multimedia) require minimum amount of bandwidth to be "effective"
- other apps ("elastic apps") make use of whatever bandwidth they get

Transport service requirements of common apps

Application	Data loss	Bandwidth	Time Sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video:10kbps-5Mbps	yes, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few kbps up	yes, 100's msec
instant messaging	no loss	elastic	yes and no

Internet transport protocols services

TCP service:

- *connection-oriented:* setup required between client and server processes
- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not providing:* timing, minimum bandwidth guarantees

UDP service:

- unreliable data transfer between sending and receiving process
- does not provide: connection setup, reliability, flow control, congestion control, timing, or bandwidth guarantee

Q: why bother? Why is there a UDP?

Internet apps: application, transport protocols

Application	Application layer protocol	Underlying transport protocol
e-mail	SMTP [RFC 5321]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	HTTP (e.g. YouTube)	TCP
Internet telephony	SIP, RTP or proprietary (e.g., Skype)	UDP or TCP

Session Initiation Protocol (SIP) is a signaling **protocol** used for initiating, maintaining, modifying and terminating real-time sessions that involve video, voice, messaging and other communications applications and services between two or more endpoints on IP networks (Whatis.com). The Real-Time Transport **Protocol (RTP)** is an Internet **protocol** standard that specifies a way for programs to manage the real-time transmission of multimedia data over either unicast or multicast network services. (techtarget)

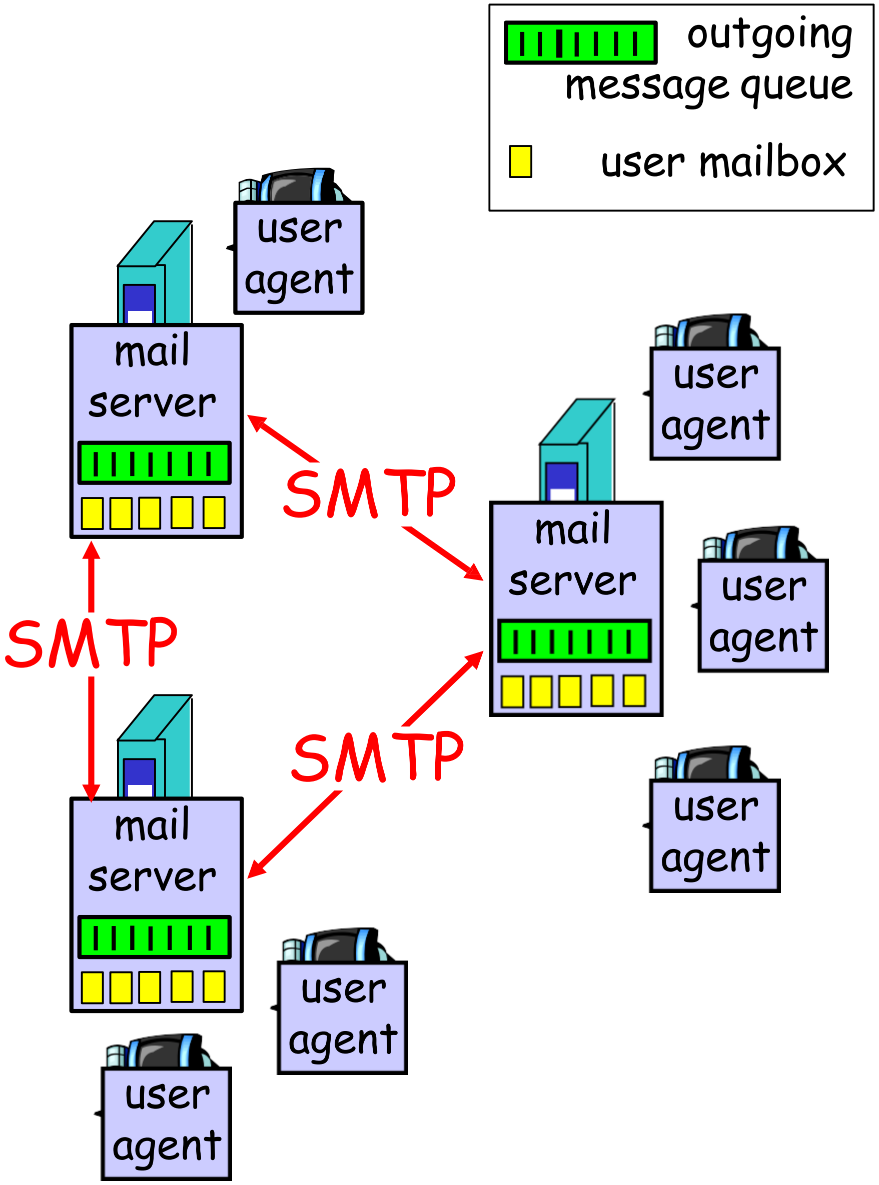
Electronic Mail

Three major components:

- user agents
- mail servers
- simple mail transfer protocol: SMTP

User Agent

- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Eudora, Outlook, elm
- outgoing, incoming messages stored on server



Sample SMTP interaction

Student: How can this be programmed? Give ideas.

At start, telnet to server.

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250  Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C:   How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

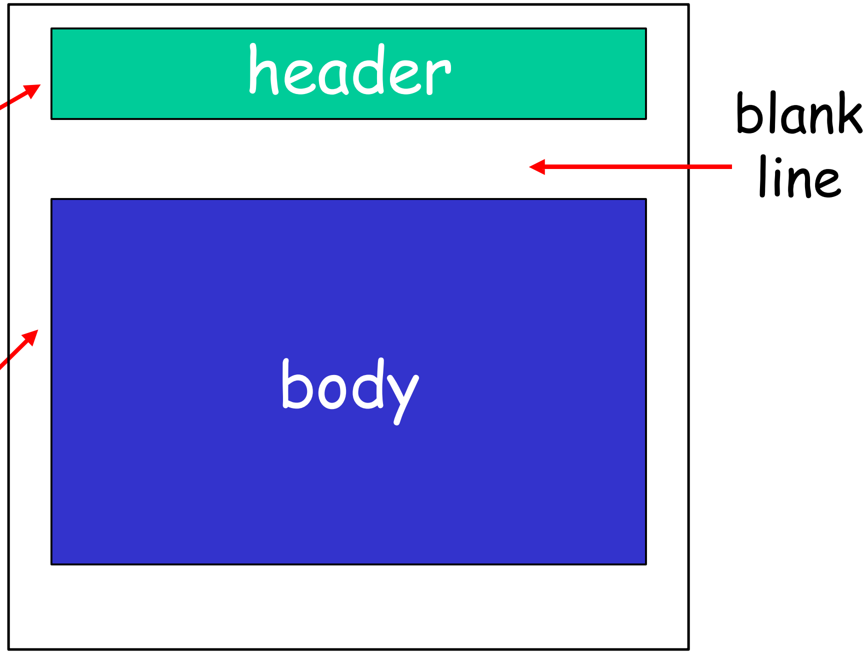
Mail message format

SMTP: protocol for exchanging email msgs

RFC 822: standard for text message format (so that receiving UA can render the email):

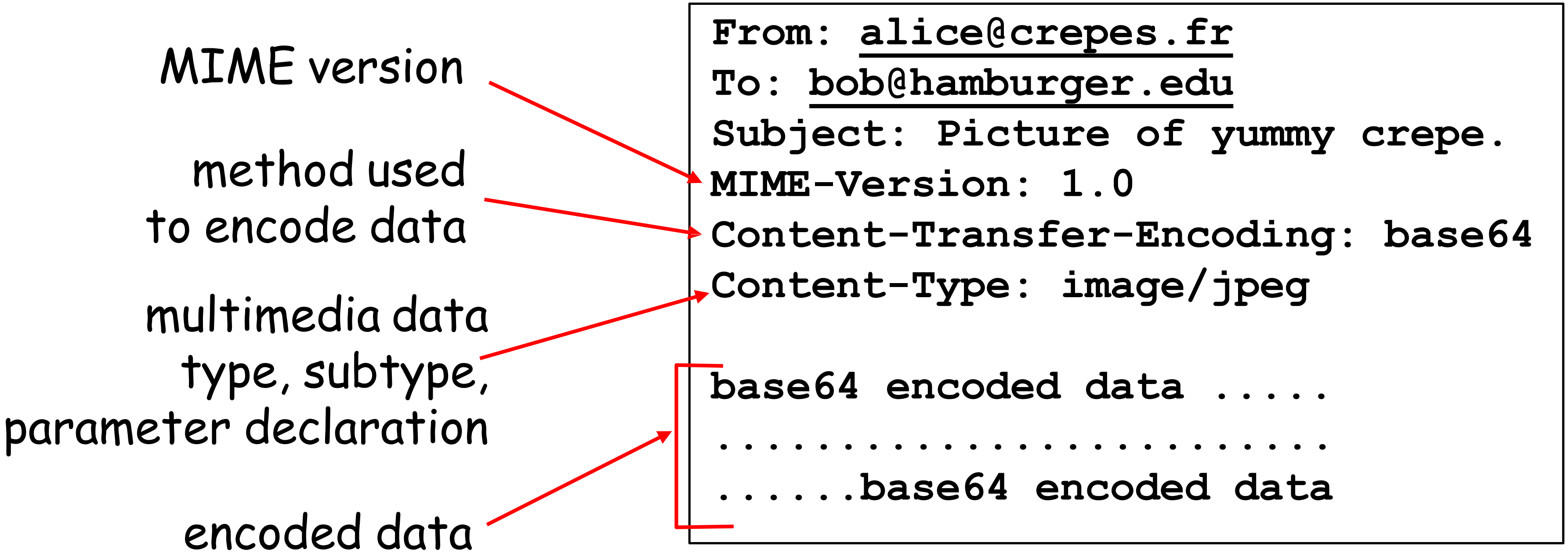
- header lines, e.g.,
  - To:
  - From:
  - Subject:

  different from SMTP commands!
- body
  - the "message", ASCII characters only



Message format: multimedia extensions

- MIME: multimedia mail extension, RFC 2045, 2056
- additional lines in msg header declare MIME content type



MIME version

method used to encode data

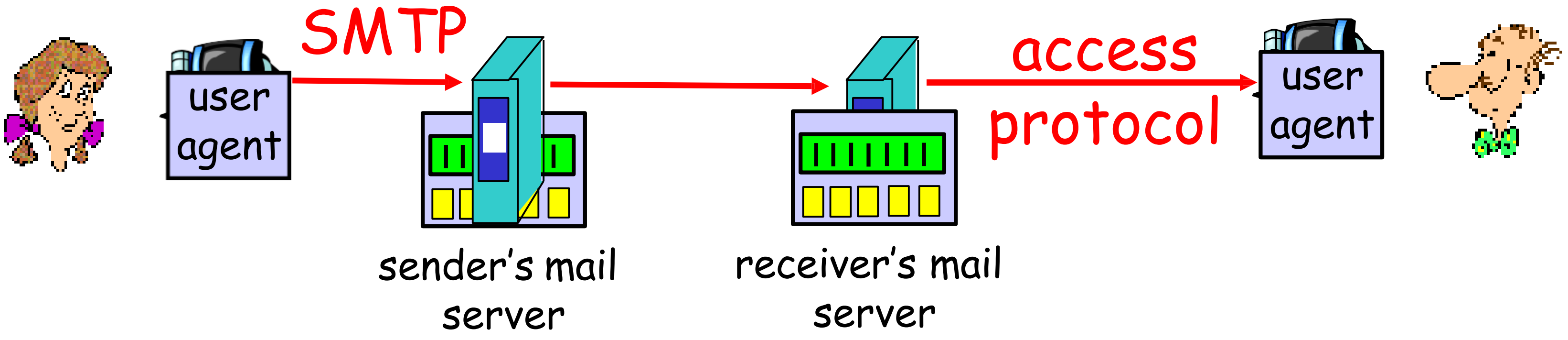
multimedia data type, subtype, parameter declaration

encoded data

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

base64 encoded data .....
.........................
......base64 encoded data
```

Mail access protocols



- SMTP: delivery/storage to receiver's server
- Mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939]
    - authorization (agent <-->server) and download
  - IMAP: Internet Mail Access Protocol [RFC 1730]
    - more features (more complex)
    - manipulation of stored msgs on server
  - HTTP: Hotmail , Yahoo! Mail, etc.

POP3 protocol

authorization phase

- client commands:
  - **user**: declare username
  - **pass**: password
- server responses
  - +OK
  - **-ERR**

transaction phase, client:

- **list**: list message numbers
- **retr**: retrieve message by number
- **dele**: delete
- **quit**

```
S: +OK POP3 server ready
C: user bob
S: +OK
C: pass hungry
S: +OK user successfully logged on
```

```
C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 2 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

POP3 (more) and IMAP

More about POP3

- Previous example uses "download and delete" mode.
- Bob cannot re-read e-mail if he changes client
- "Download-and-keep": copies of messages on different clients
- POP3 is stateless across sessions

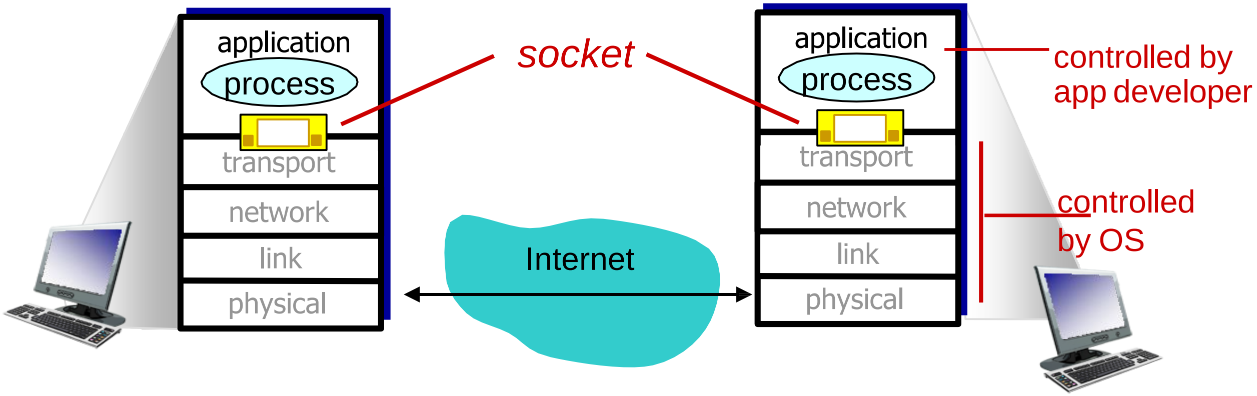
IMAP

- Keep all messages in one place: the server
- Allows user to organize messages in folders
- IMAP keeps user state across sessions:
  - names of folders and mappings between message IDs and folder name

Socket programming

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Socket programming

Two socket types for two transport services:

- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

Application Example:

1. Client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

Socket programming *with UDP*

UDP: no "connection" between client & server

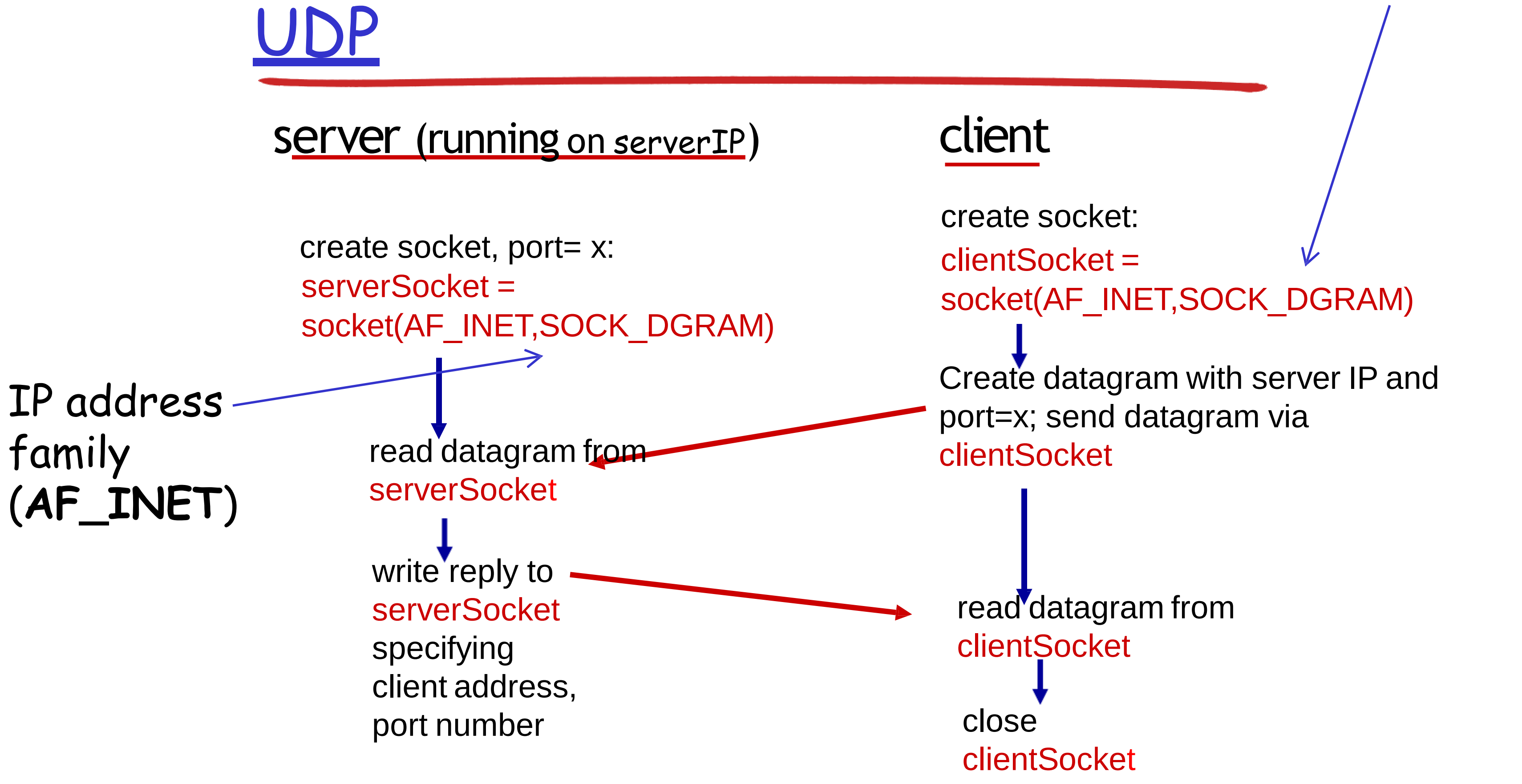
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- rcvr extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP



datagram service (SOCK_DGRAM)

server (running on serverIP)

create socket, port= x:
serverSocket =
socket(AF_INET,SOCK_DGRAM)

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

client

create socket:
clientSocket =
socket(AF_INET,SOCK_DGRAM)

Create datagram with server IP and
port=x; send datagram via
clientSocket

read datagram from
clientSocket

close
clientSocket

IP address family (AF_INET)

TutorialsPoint.com info

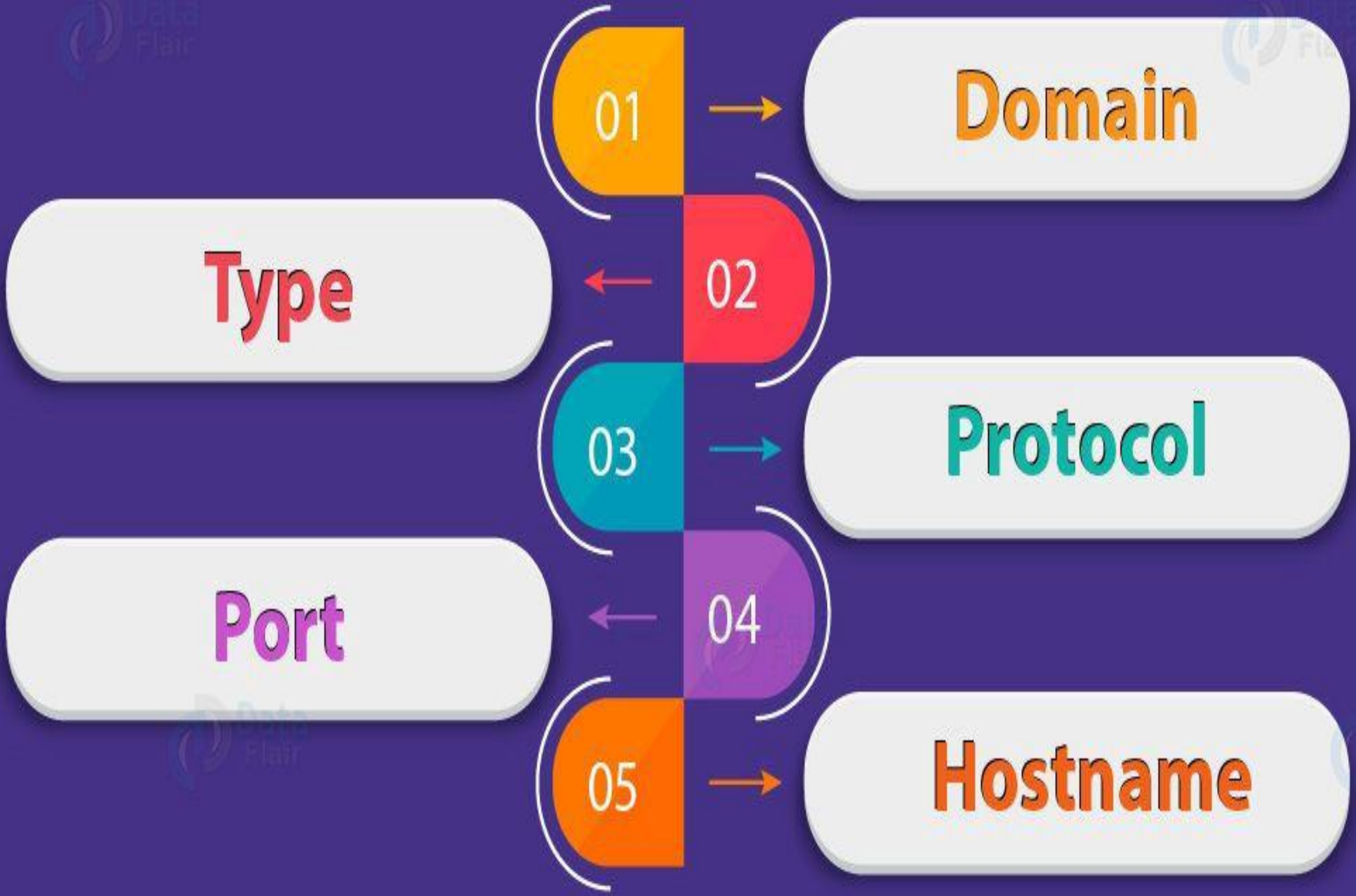
What are Sockets?

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents.

Sockets may be implemented over a number of different channel types: Unix domain sockets, **TCP, UDP**, and so on. The *socket* library provides specific classes for handling the common transports as well as a generic interface for handling the rest.

https://www.tutorialspoint.com/python/python_networking.htm

PYTHON SOCKET VOCABULARY



01 Domain

02 Type

03 Protocol

04 Port

05 Hostname

TutorialsPoint.com info

1
Domain

The family of protocols that is used as the transport mechanism. These values are constants such as **AF_INET**, PF_INET, PF_UNIX, PF_X25, and so on.

2
type

The type of communications between the two endpoints, typically **SOCK_STREAM** for connection-oriented protocols and **SOCK_DGRAM** for connectionless protocols.

3
protocol

Typically zero, this may be used to identify a variant of a protocol within a domain and type.

TutorialsPoint.com info

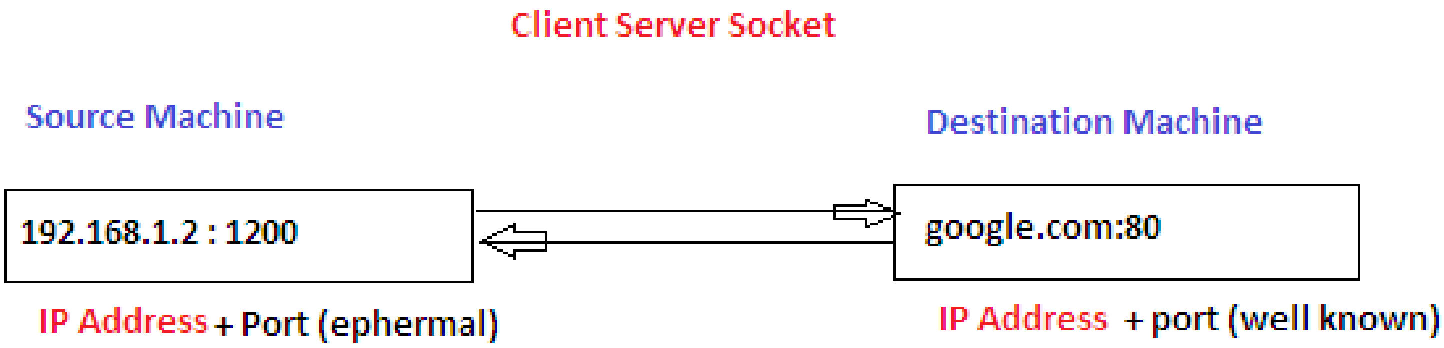
4
hostname

The identifier of a network interface

TutorialsPoint.com info

5
port

Each server listens for clients calling on one or more ports.



Client Server Socket

Source Machine

192.168.1.2 : 1200

IP Address + Port (ephermal)

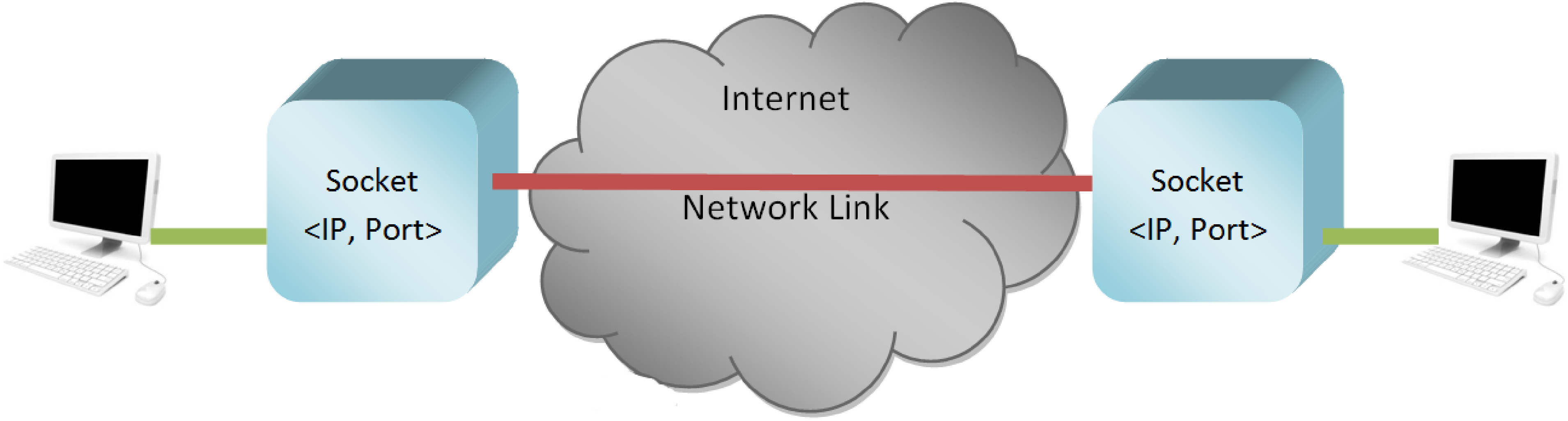
Destination Machine

google.com:80

IP Address + port (well known)

*ephermal-lasting for a short time

Socket



The socket Module

To create a socket, you must use the socket.socket() function available in socket module, which has the general syntax –

```
s = socket.socket (socket_family, socket_type, protocol=0)
```

Here is the description of the parameters –

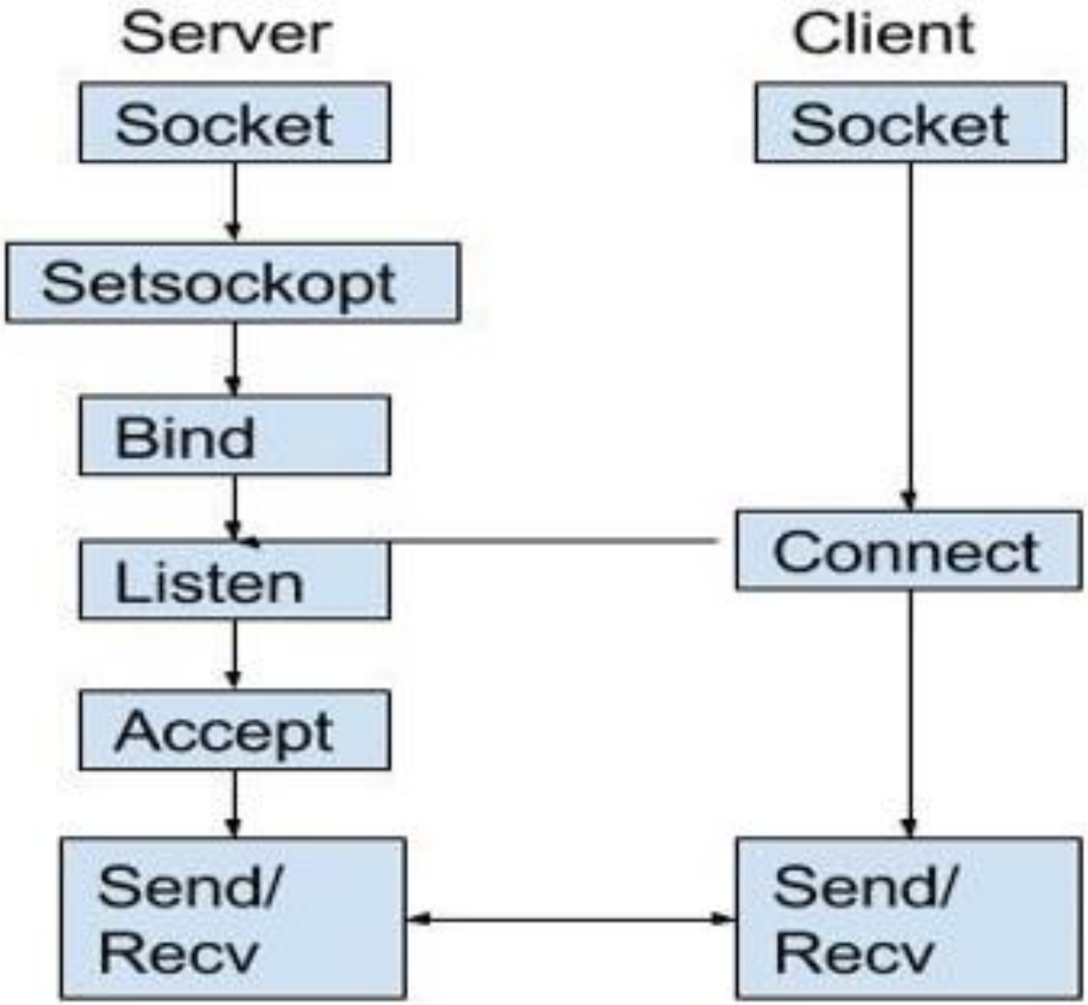
socket_family – This is either AF_UNIX or AF_INET, as explained earlier.

socket_type – This is either SOCK_STREAM or SOCK_DGRAM.

protocol – This is usually left out, defaulting to 0.

Once you have socket object, then you can use required functions to create your client or server program. Following is the list of functions required –

TCP socket diagram



TutorialsPoint.com info

Server Socket Methods (MAINLY TCP)

Sr.No. Method & Description

1

s.bind()

This method binds address (hostname, port number pair) to socket.

2

s.listen()

This method sets up and start TCP listener.

3

s.accept()

This passively accept TCP client connection, waiting until connection arrives (blocking).

TutorialsPoint.com info

Client Socket Methods

Sr.No.	Method & Description
1	**s.connect()** This method actively initiates TCP server connection.

TutorialsPoint.com info

General Socket Methods

Sr.No. Method & Description

1

s.recv()

This method receives **TCP message**

2

s.send()

This method transmits **TCP message**

3

s.recvfrom()

This method receives **UDP message**

TutorialsPoint.com info

4
s.sendto()

This method transmits **UDP message**

5
s.close()

This method closes socket

6
socket.gethostname()

Returns the hostname.

https://www.tutorialspoint.com/python/python_networking.htm

Example app: UDP client

Documentation
https://docs.python.org/

Python UDPClient

include Python's socket library

create UDP socket for server

get user keyboard input

Attach server name, port to message; send into socket

read reply characters from socket into string

print out received string and close socket

```
from socket import *
serverName = 'localhost'
serverPort = 12000
clientSocket = socket(socket.AF_INET,
                              socket.SOCK_DGRAM)
message = input('Input lowercase sentence:') #string
clientSocket.sendto(message,(serverName, serverPort))
modifiedMessage, serverAddress =
clientSocket.recvfrom(2048)
print modifiedMessage
clientSocket.close()
```

address

input([<prompt>]) If the prompt argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. socket.sendto(*string*, *address*) Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*. socket.recvfrom(*bufsize*)-Receive data from the socket. The return value is a pair (string, address) where *string* is a string representing the data received and *address* is the address of the socket sending the data.

Example app: UDP server

Python UDPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind((""  , serverPort))
print “The server is ready to receive”
while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage, clientAddress)
```

create UDP socket

bind socket to local port number 12000

loop forever

Read from UDP socket into message, getting client’s address (client IP and port)

send upper case string back to this client

socket.bind(*address*). Bind the socket to *address*.

python.org

socket.recvfrom(bufsize[, flags])

Receive data from the socket. The return value is a pair (bytes, address) where bytes is a bytes object representing the data received and address is the address of the socket sending the data.

Socket programming *with TCP*

client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

client contacts server by:

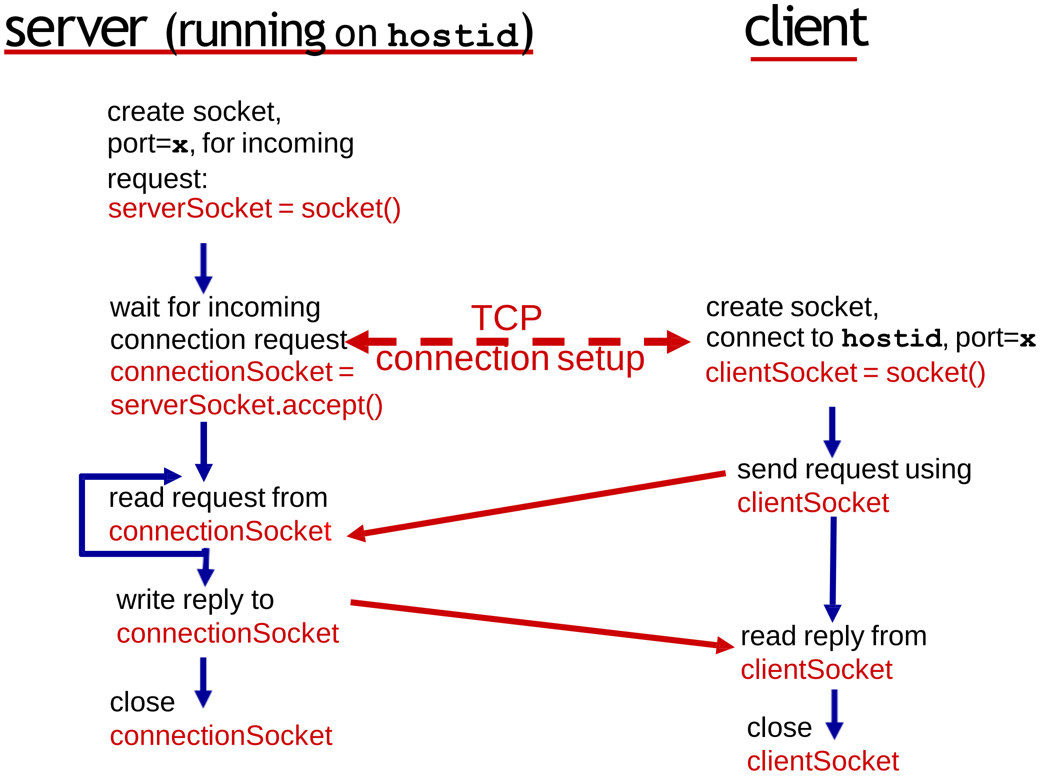
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

application viewpoint:

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

Client/server socket interaction: TCP



server (running on **hostid**)

client

create socket,
port=**x**, for incoming
request:
serverSocket = socket()

wait for incoming
connection request
connectionSocket =
serverSocket.accept()

create socket,
connect to **hostid**, port=**x**
clientSocket = socket()

send request using
clientSocket

read request from
connectionSocket

write reply to
connectionSocket

read reply from
clientSocket

close
connectionSocket

close
clientSocket

Example app:TCP client

Python TCPClient

create TCP socket for server, remote port 12000

No need to attach server name, port

```
from socket import *
serverName = ’servername’
serverPort = 12000
clientSocket = socket(AF_INET,SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = input(‘Input lowercase sentence:’)
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print ‘From Server:’, modifiedSentence
clientSocket.close()
```

socket.recv(*bufsize*) Receive data from the socket. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by *bufsize*.

Example app: TCP server

Python TCPServer

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

server waits on accept() for incoming requests, new socket created on return

read bytes from socket (but not address as in UDP)

close connection to this client (but *not* welcoming socket)

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((‘’,serverPort))
serverSocket.listen(1)
print ‘The server is ready to receive’
while 1:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
    connectionSocket.close()
```

listen(1)
-the length of the backlog queue

Other info

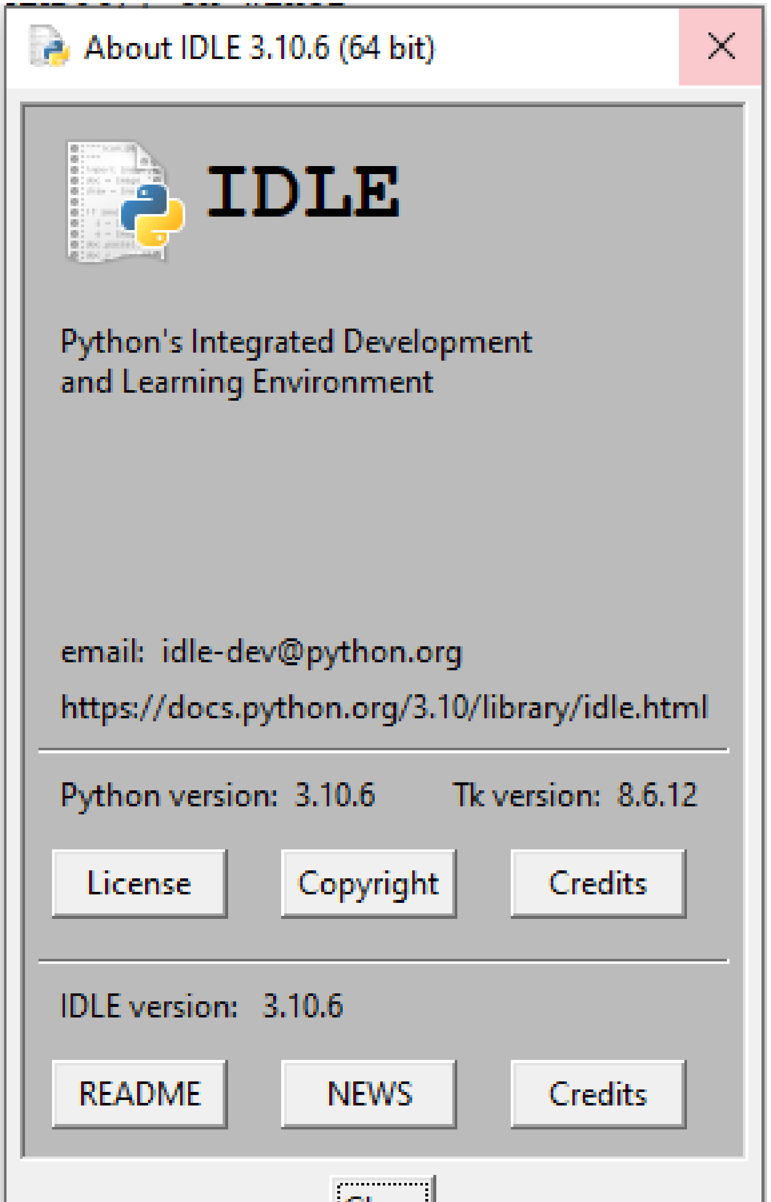
- Demo python UDP client/server
- Demo python TCP client/server
- Lecture exercise (networking problem)+solution and demo

- Students to study/research materials and use My Elearning references

  Assignment with extra time

- Tutorials to reinforce concepts
- *Lecture code to be provided on My Elearning. All students therefore have starting code for client/server programming.
- *Research is definitely necessary while programming python solutions

Software



About IDLE 3.10.6 (64 bit)

IDLE

Python's Integrated Development
and Learning Environment

email: idle-dev@python.org

https://docs.python.org/3.10/library/idle.html

Python version: 3.10.6   Tk version: 8.6.12

IDLE version: 3.10.6

Version may not be exact

You need to learn to write Python network Programming code