ICDS Spring 2025

Networking, and GUI Programming

Introduction to Network Technologies

Logistics

- **Quiz 2 (6%) @S309**
  - Recitation 003 & 005: Thursday, April 24 9:45-11:35AM
  - Recitation 004 & 006: Friday, April 25 9:45-11:35AM

- **About Final Project**
  - **Group project! 2-3 students per team**
    - Register with your teammates on Brightspace **by April 28**
  - Built upon UP 1, 2, 3
    - You will have **3 weeks to complete** the project; **video due by May 17**
    - Grading criteria to be released at Quiz 2 recitation
    - Last week's lecture & recitation as working hours for the project (May 13-16)

Agenda

- **Communication** (Socket)
  - Network fundamentals: type, topology, the Internet
  - Socket model
  - Protocols
- Event-Driven Programming
  - Callbacks
  - GUI
  - Threading (to run two tasks simultaneously in one program)

What is a network?

- A network is a linked computer system, in which data can be transferred from one machine to another.

"Body" (hardware):

- Devices (e.g., computers/ mobile phones) connected via wires
- Electrical signals (Data packets) travel on the networks

"Soul" (software):

- Protocols
- Programs built on networks; they drive the network to complete different tasks

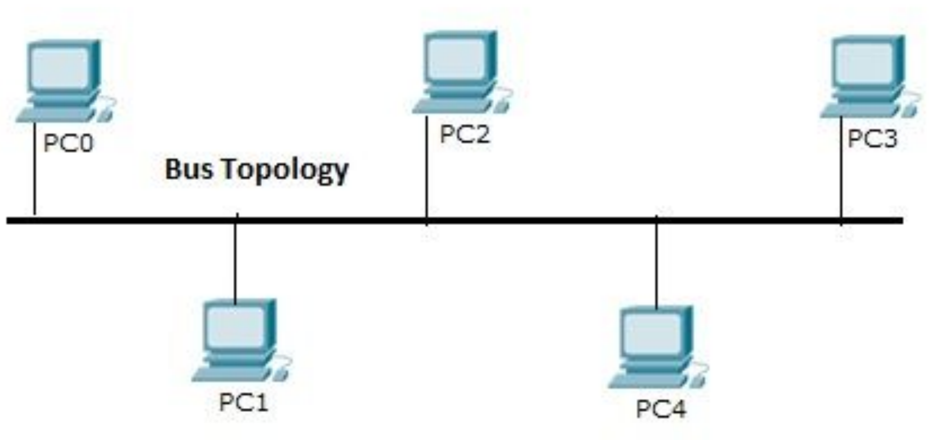
Types of networks

- Based on the range:
    - **LAN**: local area network, e.g., a collection of computers in one building
    - **WAN**: wide area network, e.g., a network links machines over greater distance such as city-to-city networks
    - **PAN**: personal area network, e.g., short range communication such as earphone connected via bluetooth

- Based on the accessibility:
    - **Open network**: operations are based on the public domain
    - **Closed network**: operations are based on a domain owned and controlled by individuals or corporations.
        - e.g., users have to pay license fees

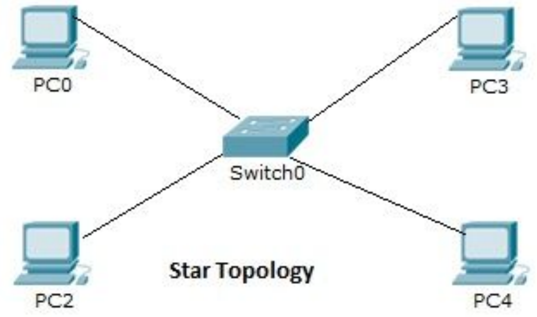
Network topologies

- Bus: machines are all connected to a common line called bus.

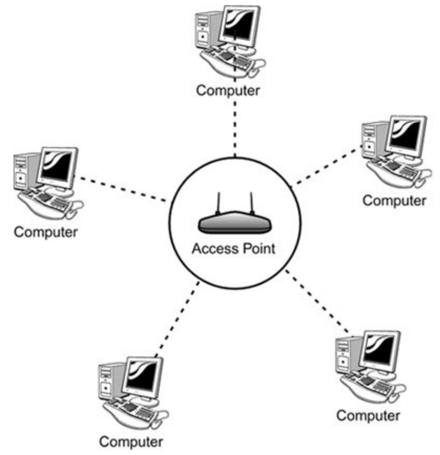


Network topologies

- Star: one machine serves as a central focal point to which all the others a connected.

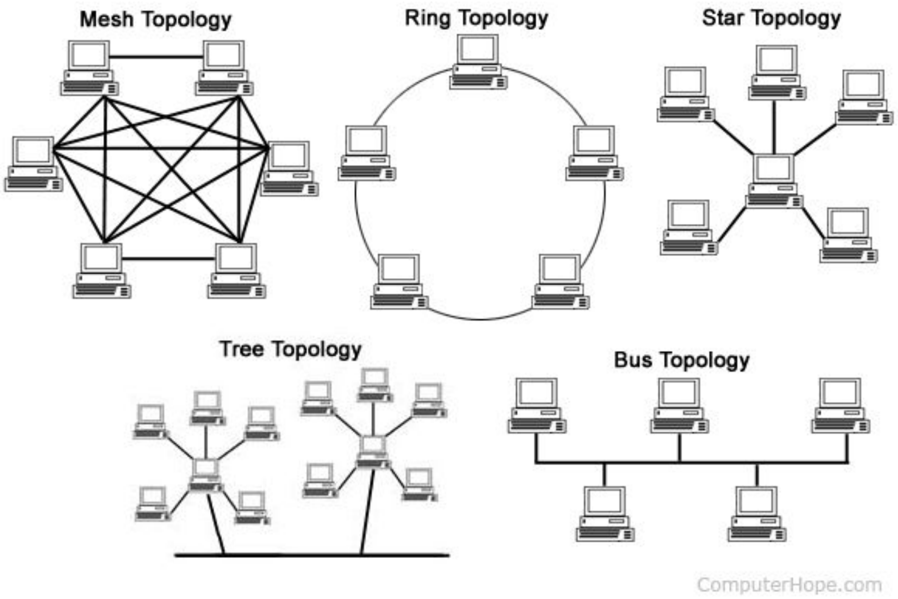


- Today, star topology is popular in wireless networks where communication is conducted by means of radio broadcast, and the central machine, called access point (AP).



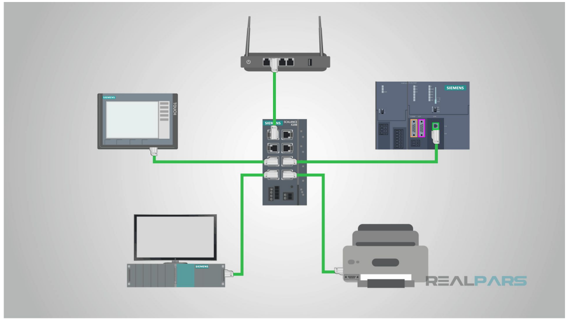
Other topologies

- Variations of bus and star topologies

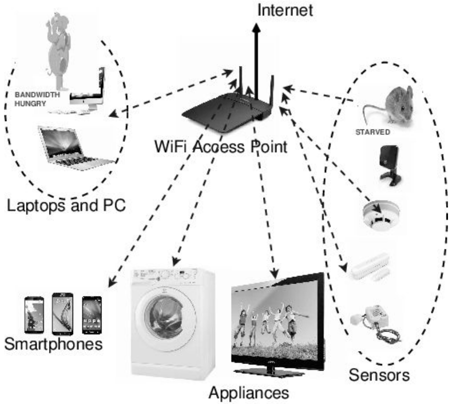


Network Connections (Connection Interface)

- Ethernet: a network where all devices are connected by cables; signals transmit via cables.



- Wi-Fi: a network where data transmits via wireless signals, often implemented in star topology.

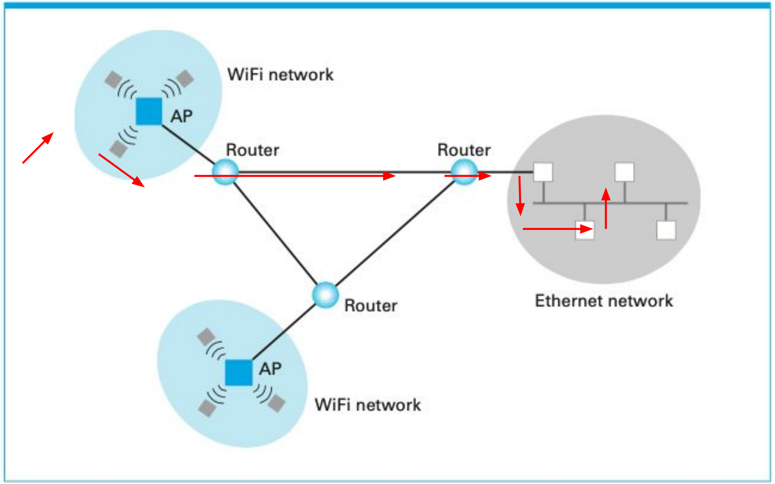


Combining networks

We connect existing networks to form an extended communication system.

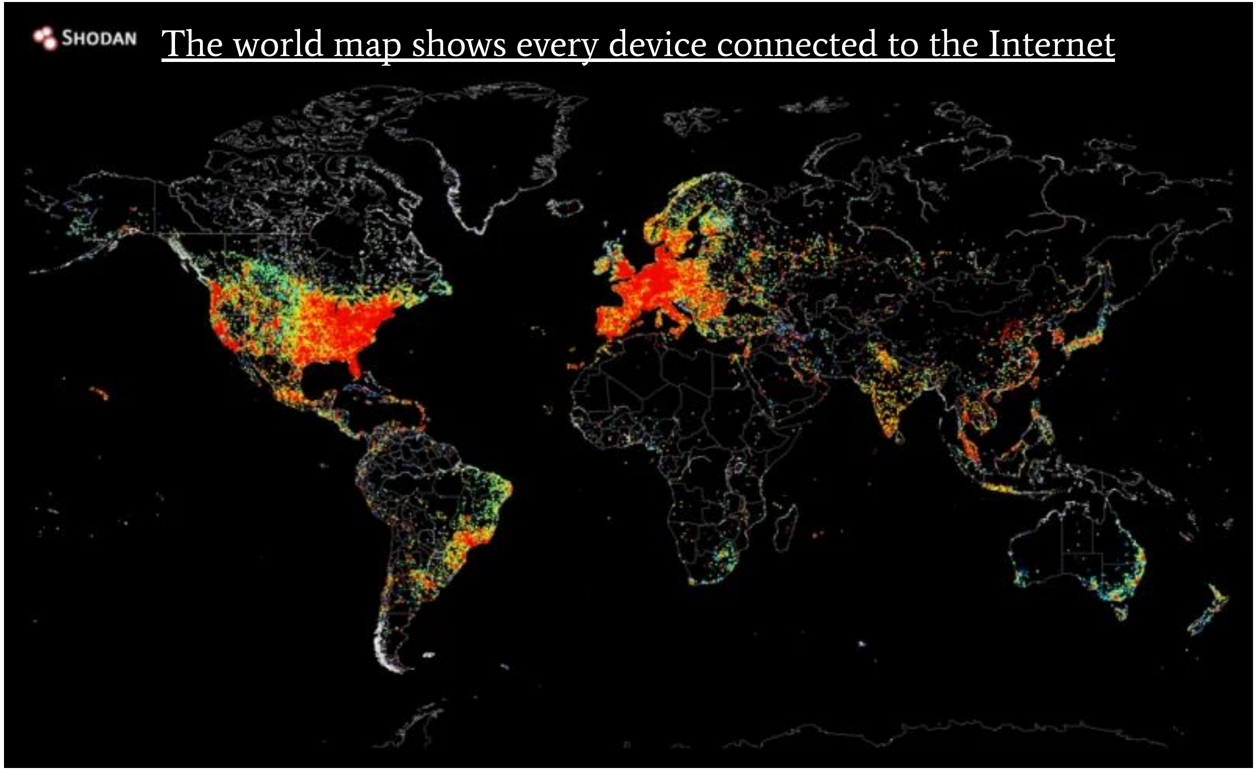
- Routers connect different networks even they have incompatible characteristics, e.g., networks with different connections such as WiFi and Ethernet.

- A network of network is called an **internet.**
    - We need an internet to connect incompatible networks.



The Internet

The Internet is an example of an internet.



How can we manipulate the networks?

Connecting computers is the first step; to make it become a functional network, we need to program it.

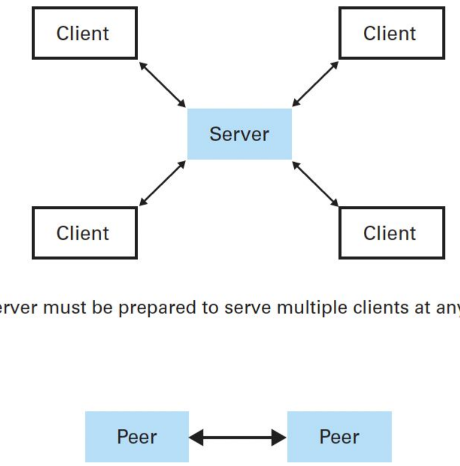
Program framework:

- Client-Server model
- Peer-To-Peer model

The above frameworks are realized by:

- **The socket model**
- **Protocols**

Figure 4.6 The client/server model compared to the peer-to-peer model



a. Server must be prepared to serve multiple clients at any time.

b. Peers communicate as equals on a one-to-one basis.

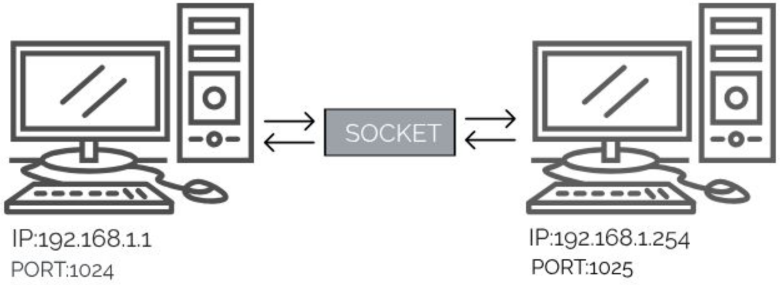
The socket model

Network Socket:

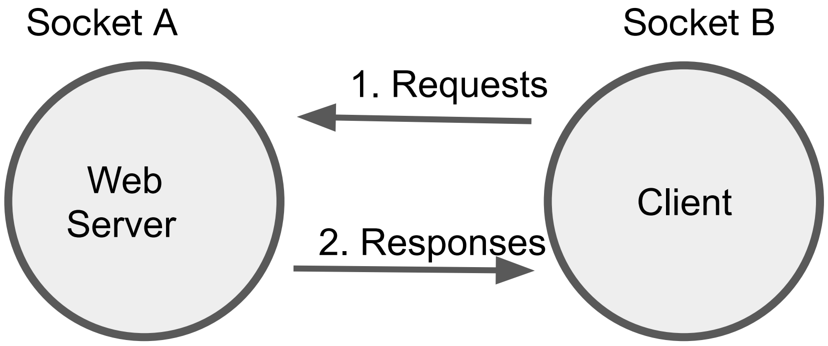
- a software structure for linking devices in a network;
- it defines a programming **interface** that helps people manipulate the network.

In the socket model,

- Each device in the network is represented by a socket with a unique IP address
- Data is transferred between sockets (socket communication)



The Web: a system built on the socket and client-server model



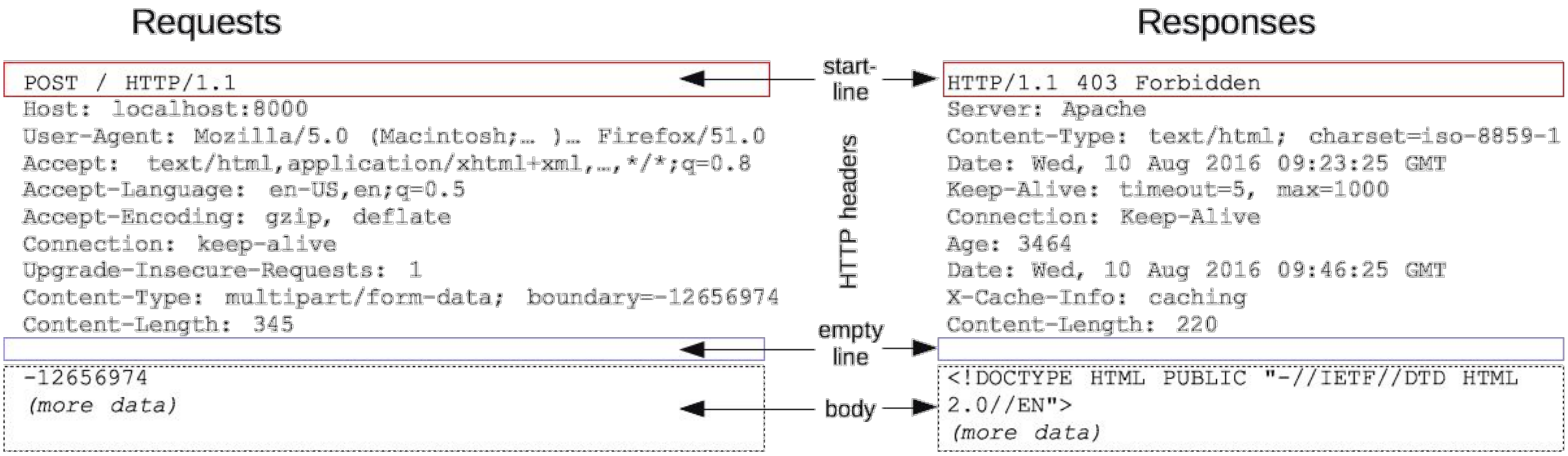
Servers and clients are considered as sockets that are connected by a network, e.g., the Internet.

- Servers and clients (web browsers) are all represented by sockets

- Communications can be simplified into two types of actions:
  - Request
  - Response

So, what should the requests/responses look like? (i.e., How can we program the requests/responses?) →We need protocols.

A real example: HTTP requests and responses

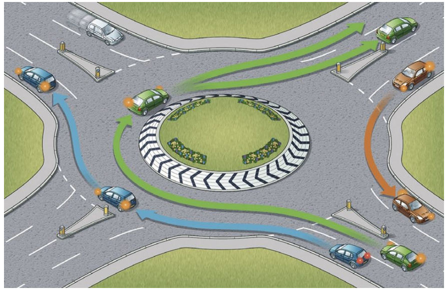


- A start line
- Headers
- Empty line
- Body

An overview of HTTP
https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#HTTP_flow

Protocols

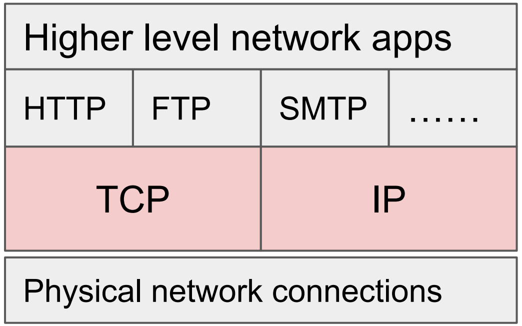
- Network protocols are rules that regulate the information exchange on a network.
  - Communication on the network is similar to human conversation.
    - When people are talking to each other, they should follow some rules, e.g., students will raise hands if they have questions.
  - Protocols like road codes.
    - There will be a traffic jam if all the computers transmit messages at the same time.



Communication protocols:

TCP/IP: low-level protocols; (work like the machine language to a computer)

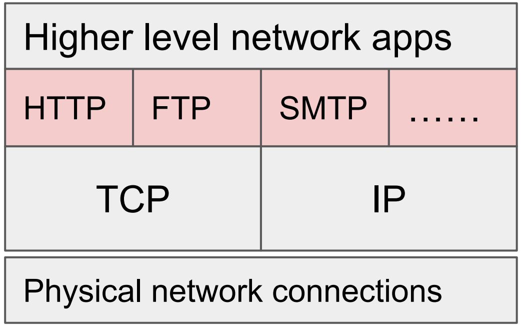
- TCP (Transmission control protocol): specifications for transmitting signals over a network (i.e., how the binary data packets being formatted and wrapped)
  - e.g., how many bits can be transmitted in a connection (it is like a specification of how many words that a sentence should have.)
- IP (Internet Protocol): specifications for assigning addresses to objects in a network



Transfer protocols

Transfer protocols (build on TCP/IP; work like high-level programming languages):

- HTTP: efficient to transfer small files like webpages
- FTP: efficient to transfer large files
- SMTP: for emails
- ……



What do they stand for?

- HTTP: HyperText Transfer Protocol
- FTP: File Transfer Protocol
- SMTP: Simple Mail Transfer Protocol

- What about HTTPS?

Your Turn

1. In the chat system you're developing, where are the communication protocol (i.e., TCP/IP) specified?
   a. client_state_machine.py
   b. chat_server.py
   c. chat_util.py

2. What is the transfer protocol used in our chat system?
   a. HTTP
   b. FTP
   c. SMTP

Scan to answer!

The TCP/IP in the chat system

In the chat system, we set up a socket that uses TCP/IP in chat_server.py,

```
#socket.AF_INET: IPv4 address family
#socket.SOCK_STREAM: a socket uses TCP protocal
self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

In chat_util.py, how do the **mysend** and **myrecv** encode/decode the messages behind?

SIZE_SPEC: the number of characters for length information.

```
def mysend(s, msg):
    #append size to message and send it
    msg = ('0' * SIZE_SPEC + str(len(msg)))[-SIZE_SPEC:] + str(msg)
    msg = msg.encode()
    total_sent = 0
    while total_sent < len(msg) :
        sent = s.send(msg[total_sent:])
        if sent==0:
            print('server disconnected')
            break
        total_sent += sent
```

In each loop, a few characters are sent. Loop will stop when all characters are sent

```
def myrecv(s):
    #receive size first
    size = ''
    while len(size) < SIZE_SPEC:
        text = s.recv(SIZE_SPEC - len(size)).decode()
        if not text:
            print('disconnected')
            return('')
        size += text
    size = int(size)
    #now receive message
    msg = ''
    while len(msg) < size:
        text = s.recv(size-len(msg)).decode()
        if text == b'':
            print('disconnected')
            break
        msg += text
    #print ('received '+message)
    return (msg)
```

It receives the length information first, then, the message.

The length of the message determines how many loops.

The transfer protocol in the chat system

We have defined a transfer protocol in the chat system already:

- It is a JSON format object (i.e., a group of (key, value), like a Python dictionary). For example,

```
"client send": { "action": "time" }
"server respond": { "action": "time", "msg": "13:40:33"}
```

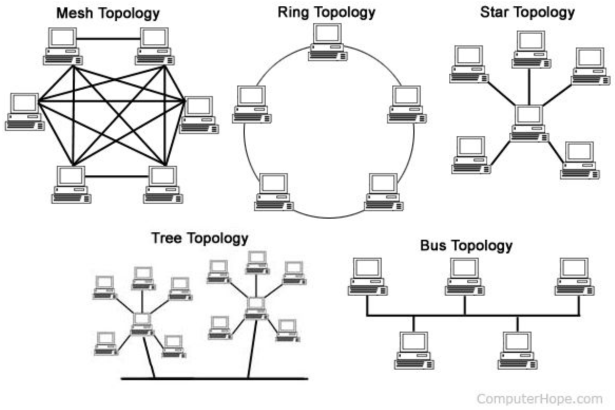
- some of the keys: “action”, “from”, “target”, “message”, etc.

```
if len(my_msg) > 0:     # my stuff going out
    mysend(self.s, json.dumps({"action":"exchange", "from":"[" + self.me + "]", "message":my_msg}))
    if my_msg == 'bye':
        self.disconnect()
        self.state = S_LOGGEDIN
        self.peer = ''
```

The messages we send and receive are all in dictionary format. We obtain the values by the corresponding keys.

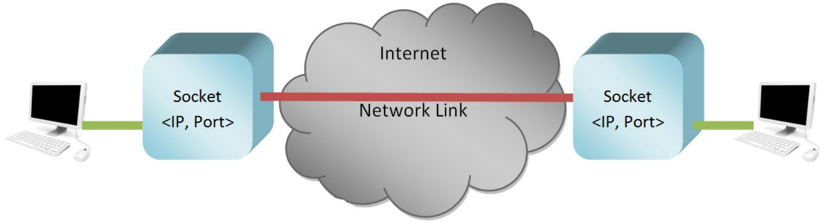
Networking: a summary

Physically connected devices



- Topology: bus, star, ...
- Connections:
  - Ethernet, WiFi, …
  - internet (the network of networks; often being connected by routers.)

The socket model



- Programming Framework:
  - Client-Server framework
  - Peer-To-Peer framework
- Foundations for creating a networking program:
  - Program interface: the socket model
  - Communication protocols
  - Transfer protocols

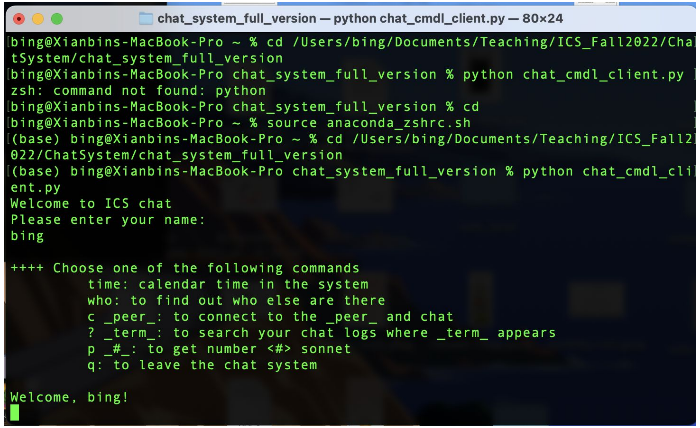
Agenda

- Communication (Socket)
  - Network fundamentals
  - Socket model
  - Protocols
- **Event-Driven Programming**
  - Callbacks
  - GUI
  - Threading (to run two tasks simultaneously in one program)

Event-Driven programming

Event-Driven Programming: a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), or message passing from other programs or threads.

- Creating responsive applications, e.g., a server, the client side, or GUI (graphical user interface)
- often involves threading



- In the chat_system, your client-side will response to you only when you input something.

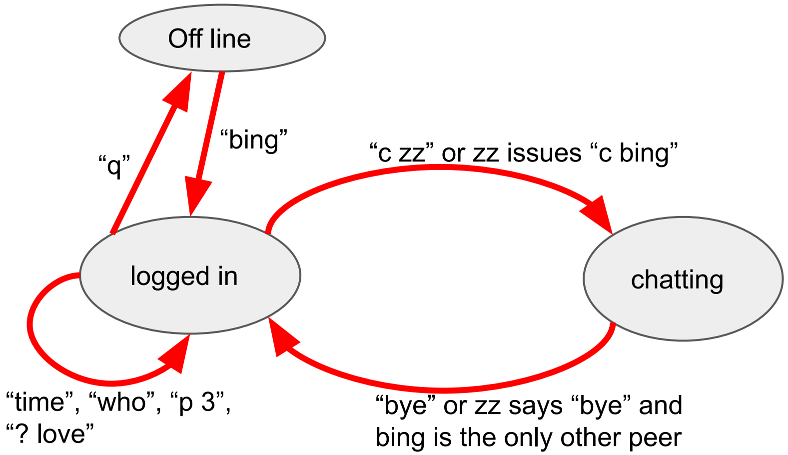
The state machine model

It is a computational model used to describe the behavior of a system in terms of its states, transitions between states, and the events that trigger these transitions.

- It's commonly used in software engineering and computer science to model and analyze systems with discrete, sequential behavior.

In a state machine model,

- **State:** Represents a condition or mode of the system at a particular point in time. The system can be in one state at a time.
- **Event:** Represents an occurrence or input that triggers a transition from one state to another.
- **Transition:** Represents a change of state in response to an event. Transitions define the conditions under which they can occur and the actions associated with them.



In the chat system,

- States: off line, logged in, and chatting.
- Event: many events,
  - e.g., when "bing" is in the state of "logged in", if he inputs "c zz", then, his state will change to chatting. If he inputs "Time", or some other strings, then, his state will remain logged in.
- Transitions: changing the self.state from one state to another.

a “responsive” app = loops + state machine

chat_cmdl_client.py

```python
from chat_client_class import *

def main():
    import argparse
    parser = argparse.ArgumentParser(description='chat client
argument')
    parser.add_argument('-d', type=str, default=None, help='server
IP addr')
    args = parser.parse_args()

    client = Client(args)
    client.run_chat()

main()
```

chat_client_class.py

```python
    def run_chat(self):
        self.init_chat()
        self.system_msg += 'Welcome to ICS chat\n'
        self.system_msg += 'Please enter your name: '
        self.output()
        while self.login() != True:
            self.output()
        self.system_msg += 'Welcome, ' + self.get_name() + '!'
        self.output()
        while self.sm.get_state() != S_OFFLINE:
            self.proc()
            self.output()
            time.sleep(CHAT_WAIT)
        self.quit()

#==============================================================
=============
# main processing loop
#==============================================================
=============
    def proc(self):
        my_msg, peer_msg = self.get_msgs()
        self.system_msg += self.sm.proc(my_msg, peer_msg)
```

chat_state_machine.py

```python
    def proc(self, my_msg, peer_msg):
        self.out_msg = ''
#==============================================================================
# Once logged in, do a few things: get peer listing, connect, search
# And, of course, if you are so bored, just go
# This is event handling instate "S_LOGGEDIN"
#==============================================================================
        if self.state == S_LOGGEDIN:
            # todo: can't deal with multiple lines yet
            if len(my_msg) > 0:

                if my_msg == 'q':
                    self.out_msg += 'See you next time!\n'
                    self.state = S_OFFLINE

                elif my_msg == 'time':
                    mysend(self.s, json.dumps({"action":"time"}))
                    time_in = json.loads(myrecv(self.s))["results"]
                    self.out_msg += "Time is: " + time_in

                elif my_msg == 'who':
                    mysend(self.s, json.dumps({"action":"list"}))
                    logged_in = json.loads(myrecv(self.s))["results"]
                    self.out_msg += 'Here are all the users in the system:\n'
```

In the Unit Project, we use while loops, and a state machine model to make responses.

- Different states will have different responses.

A GUI resembles a state machine


```
while True:
    if condition_1 is True:
        run some function
    elif condition_2 is True:
        run some function
    else condition_3 is True:
        run some function
```

Those functions are called callback functions.

- The window is alway running, like a "`while True:`" loop;
- Each button is a condition;
- When the mouse click on one of the button ⇒ a specific event occurs ⇒ some condition is True ⇒ run the code in the branch of that condition (e.g., convert Fahrenheit to Celsius)

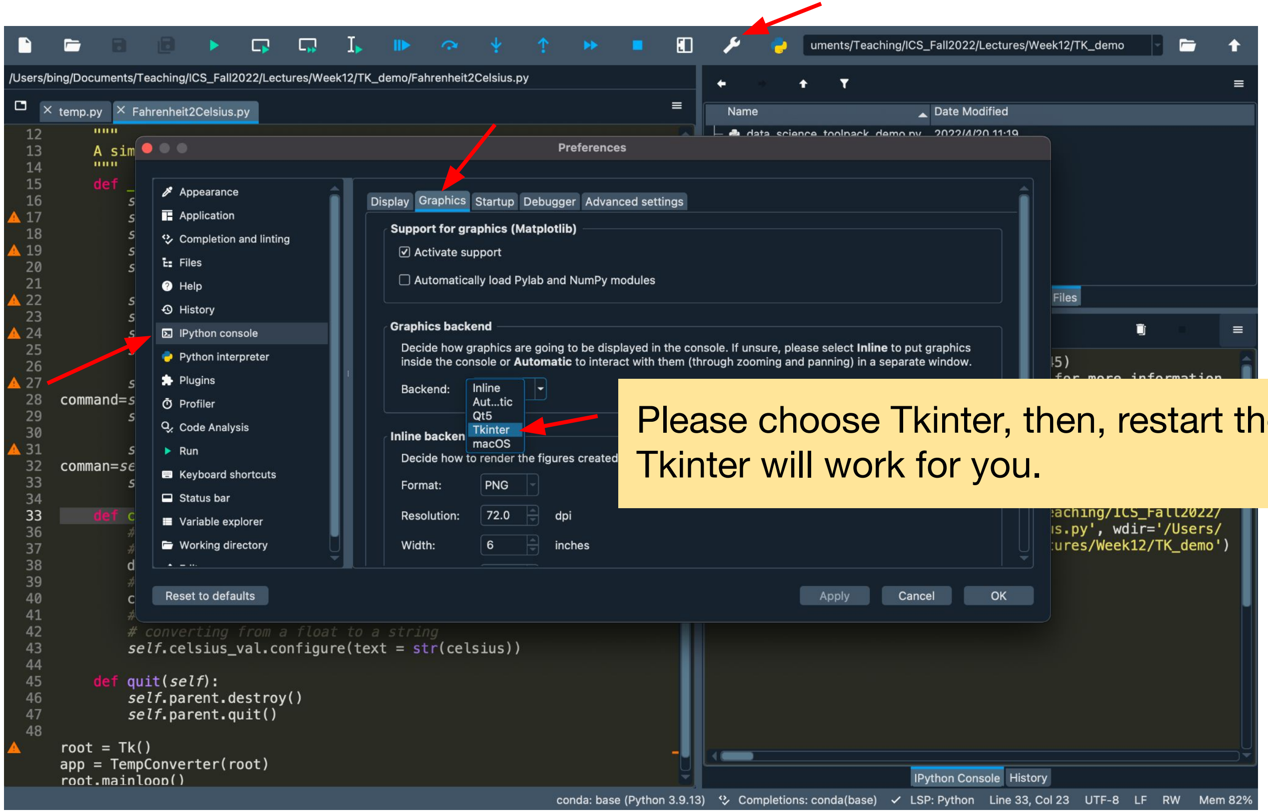
Tkinter (a Python GUI framework)

- Tkinter (Tk interface) is Python’s default GUI toolkit, although there are many others.
- It is reasonably simple and lightweight and should be perfectly adequate for GUI programs of small to medium complexity.

If you would like to know more about Tkinter,

- Tkinter **demo & lecture notes** at **Brightspace/Lectures/Week 11/**
- **Chapter 13**, GUI programming, in Starting out with Python (i.e., the Gaddis text).

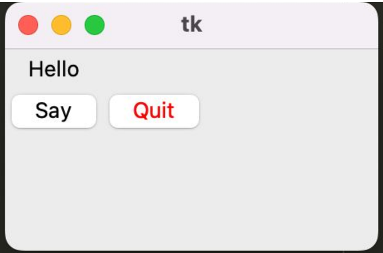
Settings of Spyder for Running Tkinter



Please choose Tkinter, then, restart the console.
Tkinter will work for you.

Hello world

```python
"""

from tkinter import *

class HelloWorld:

    def __init__(self, parent):
        self.label =  Label(parent, text="Hello")
        self.label.grid(column=0, row=0)
        self.hello_botton = Button(parent, text="Say", command=self.say_hi)
        self.hello_botton.grid(column=0, row=1)
        self.qui_button = Button(parent, text="Quit", fg="red", command=parent.destroy)
        self.qui_button.grid(column=1, row=1)

    def say_hi(self):
        print("Hi there")


root = Tk()
app = HelloWorld(root)
root.mainloop()
```



Callback functions: they will be executed when the event occurs.

1. We create a root window
2. then, assign the interface to it
3. call the `.mainloop()` to start the root window

Tkinter program: two parts

```
from tkinter import *
class ClassName:
    def __init__(self, parent):
    # code to create and display widgets goes here as
    # well as any other variables needed for the class
    def some_callback_method(self):
    # code you want to execute in response to an event
    # goes here
    def maybe_another_callback(self):
    # maybe code to respond to a different event
    # goes here
    # main routine
root = Tk()
instance_name = ClassName(root)
root.mainloop()
```

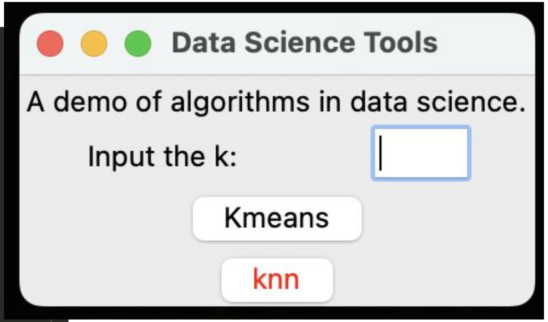
- Creating and displaying widgets
- Responding to events (callback methods)

GUI for Data Science Tools

```python
import os
import sys
sys.path.append(os.getcwd()+'/kmeans')
sys.path.append(os.getcwd()+'/knn')
sys.path.append(os.getcwd()+'/data')

import random
from tkinter import *
from kmeans import kmeans
from knn import knn


class Tools():

    def __init__(self, parent):
        self.label =  Label(parent, text="A demo of algorithms in data science.")
        self.label.grid(row=0, columnspan=2)

        # self.samples = self.generate_data()

        self.k = Entry(parent, width=4)
        self.k.grid(column=1, row=1)
        self.k_label = Label(parent, text="Input the k:")
        self.k_label.grid(column=0, row=1)

        self.kmeans_botton = Button(parent, text="Kmeans", command=self.kmeans)
        self.kmeans_botton.grid(columnspan=2, row=3)

        self.knn_button = Button(parent, text="knn", fg="red", command=self.knn)
        self.knn_button.grid(columnspan=2, row=4)
```

Import packages:

- Kmeans / knn we made in-class
- Recall Recitation Week3/Building a package



Button for kmean

Button for knn

Methods

```
    def kmeans(self):
        print("kmeans")
        k = int(self.k.get())
        kmeans.run_kmeans_on_iris(k)

    def knn(self):
        print("knn")
        k = int(self.k.get())
        knn.run_knn_on_iris(k)
```

- There are different ways to integrate the kmeans and knn into the GUI.
- In this example, we simply import the kmeans and knn as packages
- In kmeans/knn, some modifications are needed which adapt the code to the GUI.

Modification in k-means/k-nn

- Firstly, adding a __init__.py in the folder of K-means/k-nn;
- Secondly, change the path of the import statements in kmeans.py and knn.py;

> - `from knn.sample import` ...: this guarantees the imported files are from the knn folder

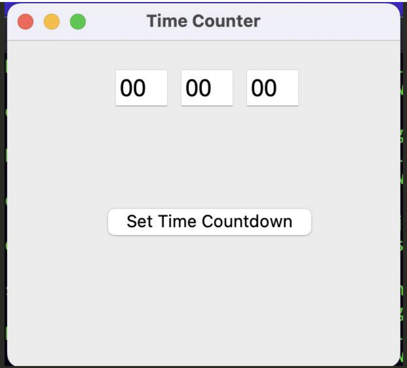
```python
from knn.sample import Sample, plot_samples
```

```python
def run_kmeans_on_iris(k):
    f = open('./data/iris.csv', 'r')
    raw_data = f.readlines()
```

> - Apart from the mentioned changes, there are a few modifications in the existing code, which makes it easier for the GUI to call functions in the packages. Please refer to the shared code in the Brightspace/Lectures/Week 12/
> - The key to integrate different programs is the interface, which allows the arguments to be transmitted.

Running two tasks simultaneously

```python
import time
from tkinter import *
from tkinter import messagebox
import threading


class Timer():
    def __init__(self, parent):
        self.parent = parent
        self.hour = StringVar()
        self.minute = StringVar()
        self.second = StringVar()
        self.hour.set("00")
        self.minute.set("00")
        self.second.set("00")

        self.hourEntry = Entry(parent, width=3, font=("Arial", 18, ""), textvariable=self.hour)
        self.hourEntry.place(x=80, y=20)
        self.minuteEntry = Entry(parent, width=3, font=("Arial", 18, ""), textvariable=self.minute)
        self.minuteEntry.place(x=130, y=20)
        self.secondEntry = Entry(parent, width=3, font=("Arial", 18, ""), textvariable=self.second)
        self.secondEntry.place(x=180, y=20)
        self.start_button = Button(parent, text='Set Time Countdown', bd='5',
                             command= self.start)
        self.start_button.place(x = 70, y = 120)
```



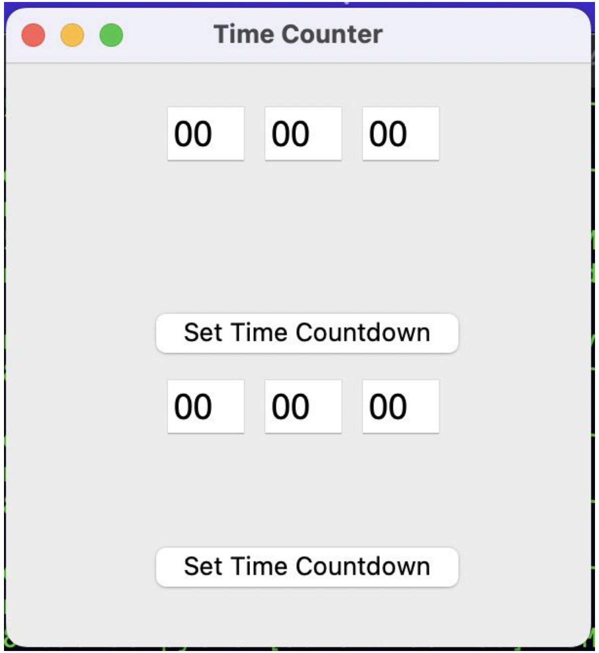
A timer counts down the time when clicking the button (which calls the self.start)

self.start

```
    def start(self):
        try:
            # the input provided by the user is
            # stored in here :temp
            temp = int(self.hour.get())*3600 + int(self.minute.get())*60 + int(self.second.get())
        except:
            print("Please input the right value")

        while temp >-1:

            # divmod(firstvalue = temp//60, secondvalue = temp%60)
            mins,secs = divmod(temp, 60)

            # Converting the input entered in mins or secs to hours,
            # mins ,secs(input = 110 min --> 120*60 = 6600 => 1hr :
            # 50min: 0sec)
            hours=0
            if mins >60:

                # divmod(firstvalue = temp//60, secondvalue
                # = temp%60)
                hours, mins = divmod(mins, 60)

            # using format () method to store the value up to
            # two decimal places
            self.hour.set("{0:2d}".format(hours))
            self.minute.set("{0:2d}".format(mins))
            self.second.set("{0:2d}".format(secs))

            # updating the GUI window after decrementing the
            # temp value every time
            self.parent.update()
            time.sleep(1)

            # when temp value = 0; then a messagebox pop's up
            # with a message:"Time's up"
            if (temp == 0):
                messagebox.showinfo("Time Countdown", "Time's up ")

            # after every one sec the value of temp will be decremented
            # by one
            temp -= 1
```

The loop will not stop when temp > -1

Timer counters



- When Timer 1 starts, it will count until the self.start stops (i.e., when temp <= -1).
- Timer 2 will start only after Timer 1 stops.

Can the two timers work together?

Using threading

Python threading allows you to have different parts of your program run simultaneously and can simplify your design.

```
def proc_start(self):
    process = threading.Thread(target=self.start)
    process.daemon = True
    process.start()
```

- `.daemon = True`: the thread will stop immediately when the program exits.

- Import the threading package
- Create a thread for self.start

By calling self.proc_start, a thread will be created to run self.start (you can image a thread is another computer/cpu, and the function runs on it);
So, the main program will go on with the following statements.

Note: not all functions/packages support threading, e.g, some functions in matplotlib collapse when using threading.

Modify the start

```python
    def start(self, hour, minute, second):
        try:
            # the input provided by the user is
            # stored in here :temp
            temp = int(hour.get())*3600 + int(minute.get())*60 + int(second.get())
        except:
            print("Please input the right value")

        while temp >-1:
            # divmod(firstvalue = temp//60, secondvalue = temp%60)
            mins,secs = divmod(temp, 60)
            # Converting the input entered in mins or secs to hours,
            # mins ,secs(input = 110 min --> 120*60 = 6600 => 1hr :
            # 50min: 0sec)
            hours=0
            if mins >60:
                # divmod(firstvalue = temp//60, secondvalue
                # = temp%60)
                hours, mins = divmod(mins, 60)
            # using format () method to store the value up to
            # two decimal places
            hour.set("{0:2d}".format(hours))
            minute.set("{0:2d}".format(mins))
            second.set("{0:2d}".format(secs))
            # updating the GUI window after decrementing the
            # temp value every time
            self.parent.update()
            time.sleep(1)
            # when temp value = 0; then a messagebox pop's up
            # with a message:"Time's up"
            if (temp == 0):
                messagebox.showinfo("Time Countdown", "Time's up ")
            # after every one sec the value of temp will be decremented
            # by one
            temp -= 1
```

Define a thread

```
def process_start(self, hour, minute, second):
    process = threading.Thread(target=lambda : self.start(hour, minute, second))
    process.daemon = True
    process.start()
```

We need to use a lambda function to pass arguments to self.start in the thread because the “target” should be a function. (not a function call, so we wrap the self.start by a lambda function)

- More about the lambda function: https://www.w3schools.com/python/python_lambda.asp

Modifying the first timer in the window

```python
self.hour = StringVar()
self.minute = StringVar()
self.second = StringVar()
self.hour.set("00")
self.minute.set("00")
self.second.set("00")

self.hourEntry = Entry(parent, width=3, font=("Arial", 18, ""), textvariable=self.hour)
self.hourEntry.place(x=80, y=20)
self.minuteEntry = Entry(parent, width=3, font=("Arial", 18, ""), textvariable=self.minute)
self.minuteEntry.place(x= param parent
self.secondEntry = Entry(parent, width=3, font=("Arial", 18, ""), textvariable=self.second)
self.secondEntry.place(x=180, y=20)
self.start_button = Button(parent, text='Set Time Countdown', bd='5',
                command=lambda:self.process_start(self.hour, self.minute, self.second))
self.start_button.place(x = 70, y = 120)
```

Using the lambda function trick again, to pass arguments to self.process_start.

Creating the second timer in the window

```python
self.hour2 = StringVar()
self.minute2 = StringVar()
self.second2 = StringVar()
self.hour2.set("00")
self.minute2.set("00")
self.second2.set("00")
self.hourEntry2 = Entry(parent, width=3, font=("Arial", 18, ""), textvariable=self.hour2)
self.hourEntry2.place(x=80, y=180)
self.minuteEntry2 = Entry(parent, width=3, font=("Arial", 18, ""), textvariable=self.minute2)
self.minuteEntry2.place(x=130, y=180)
self.secondEntry2 = Entry(parent, width=3, font=("Arial", 18, ""), textvariable=self.second2)
self.secondEntry2.place(x=180, y=180)
self.start_button2 = Button(parent, text='Set Time Countdown', bd='5',
            command= lambda:self.process_start(self.hour2, self.minute2, self.second2))
self.start_button2.place(x = 70, y = 280)
```

Using the lambda function trick again, to pass arguments to self.process_start.

Homeworks

[GUI] Hands-on - Finish in-class exercises:

- KNN modification in DS toolkit: page 32-34
- Timer threading: page 37-42

[GUI] Reading - Chapter 13: GUI programming

- Brightspace/Reference Books/*Starting out with Python, by Tony Gaddis*