IEMS 5703
Network Programming and System Design

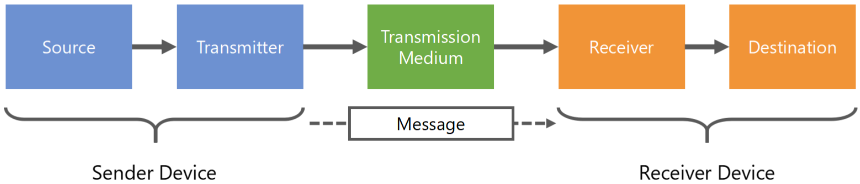
Lecture 2 - Computer Networks and Network Programming

Albert Au Yeung
18th January, 2018

Computer Networking

Data Communication

- Exchange of data between two devices using some form of transmission medium
- A simplified communication model:

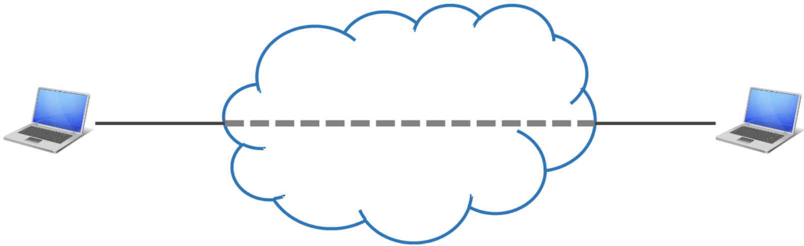


- When performing communication, we need **protocols**: rules that govern how data is transmitted in this system

Methods of Communication

Switching

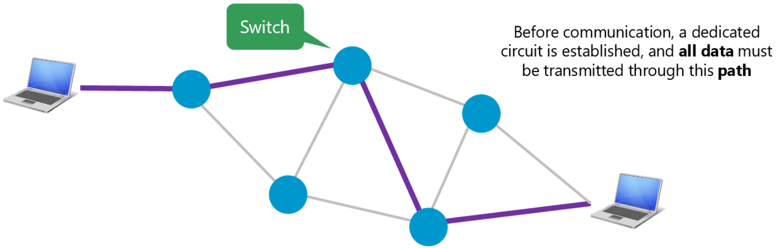
- When two computers need to communication over a network, we need to know how to connect them to each otaher



Methods of Communication

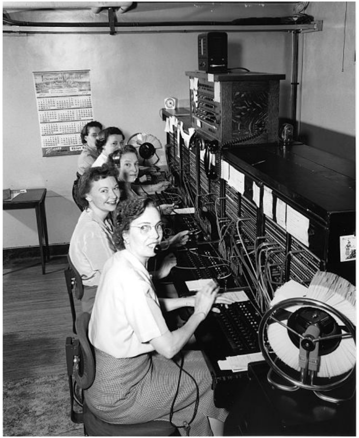
Circuit Switching

- To establish a dedicated communication link (a circuit) between two computers when they need to talk to each other
- Before communication, a dedicated circuit is established, and all data must be transmitted through this path



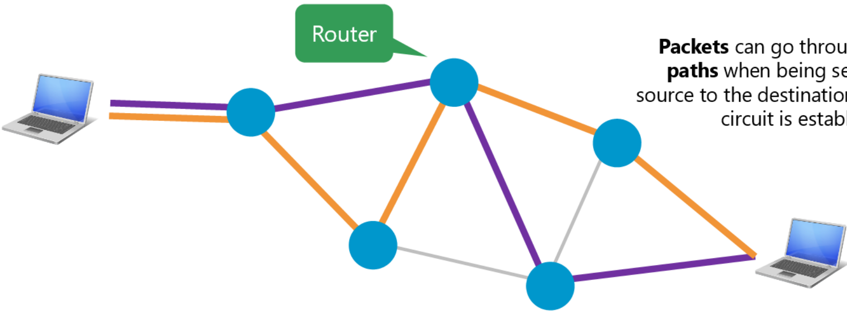
Methods of Communication

- The Analog Telephone Network as a Classic Circuit Switching Example
- Left: Telephone operators in a private branch exchange in 1952.
- (Ref: https://en.wikipedia.org/wiki/Switchboard_operator)



Methods of Communication

Packet Switching



Packets can go through **different paths** when being sent from the source to the destination, no dedicated circuit is established

Packet Switching

Advantages of Packet Switching

- **Efficiency**: The network can be used in a more efficient way
  (The same link can be shared by many different connections)
- **Reliability**: More fault tolerant
  (Consider when a switch is broken in the middle of the communication)

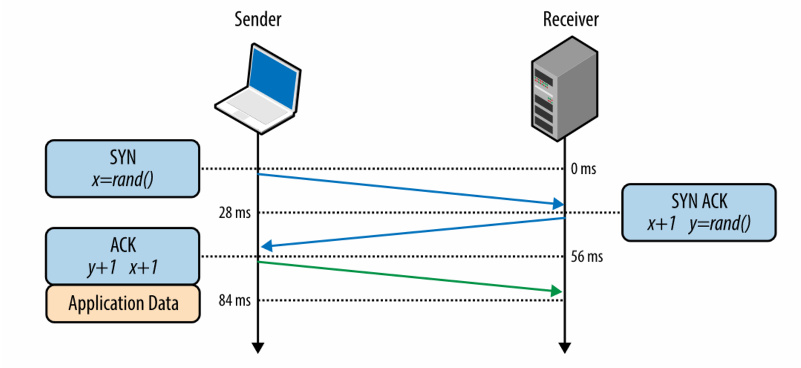
Protocols

What are protocols?

- A set of **rules** that govern how communication parties interact with each other
- An **agreement** between the communicating entities
- Two devices need to agree on common protocols when they communicate
- Things that a protocol should define:
    - The format of the **addressing scheme**
    - How do we specify the **start** and **end** of a data stream?
    - How do we **handle errors** or data loss?
    - How to handle problems in data transfer?
    - ...

Protocols

An example: TCP's three way handshake



Reference: https://hpbn.co/building-blocks-of-tcp/

Layered Architecture

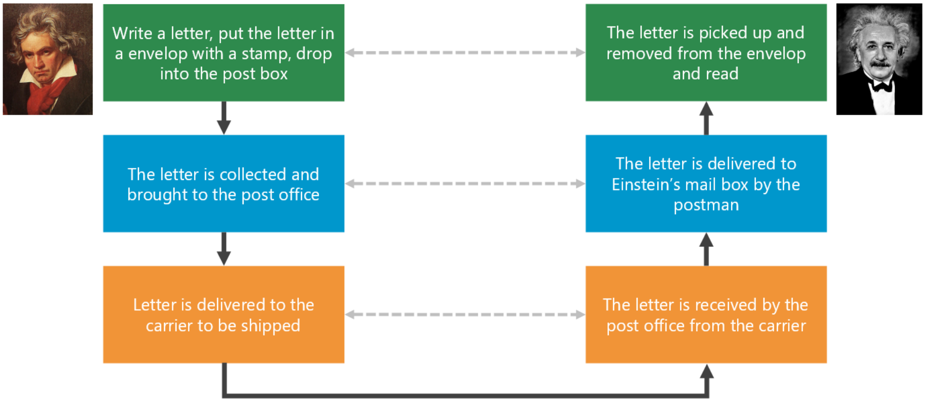
- Computer networks usually adopt a **layered architecture**
- Each layer has its own functions, and only interact directly with the one **above** and **below** it
- Protocols are used to allow entity (e.g. a program) in one host to communicate with another entity in another host on the **same** layer

Question

- Why do we want to have a *layered* architecture?
- Consider the example in the next slide

Layered Architecture

- Imagine Beethoven is writing a letter to Einstein…



Layered Architecture

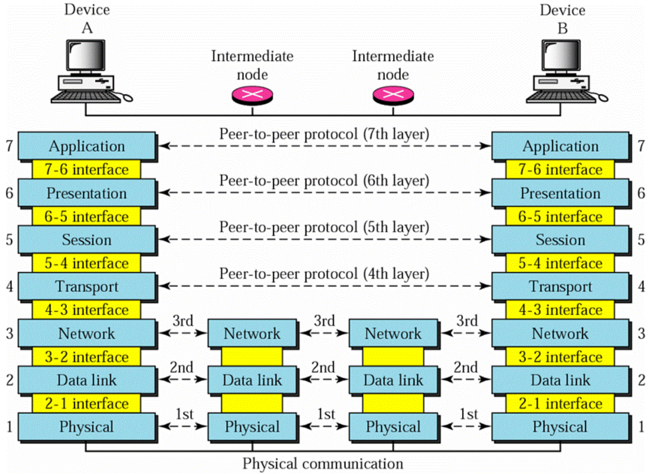
The ISO OSI (Open System Interconnection) 7-Layer Model

- A **theoretical** model of how a computer network should work
- It organises different functions of a network into **seven** different layers
- It specifies the interfaces for communication between different layers and different endpoints

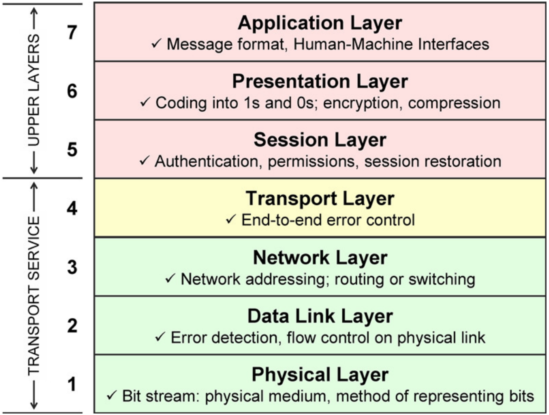
Note:

- It is a theoretical model
- It is not a program or software
- Practical networks may be implemented in a different way

OSI Model


Device A
Device B
Intermediate node
Intermediate node
7 Application
7-6 interface
6 Presentation
6-5 interface
5 Session
5-4 interface
4 Transport
4-3 interface
3 Network
3-2 interface
2 Data link
2-1 interface
1 Physical
Peer-to-peer protocol (7th layer)
Peer-to-peer protocol (6th layer)
Peer-to-peer protocol (5th layer)
Peer-to-peer protocol (4th layer)
3rd
2nd
1st
Network
Data link
Physical
Application 7
7-6 interface
Presentation 6
6-5 interface
Session 5
5-4 interface
Transport 4
4-3 interface
Network 3
3-2 interface
Data link 2
2-1 interface
Physical 1
Physical communication

OSI Model



Source: http://nhprice.com/what-is-ios-model-the-overall-explanation-of-ios-7-layers.html

- Describes **what** need to be done to enable data to be sent from one device to another device
- Each layer is a collection of similar functions
- Each layer serves the layer **above** it
- Each layer get services from the one **below** it

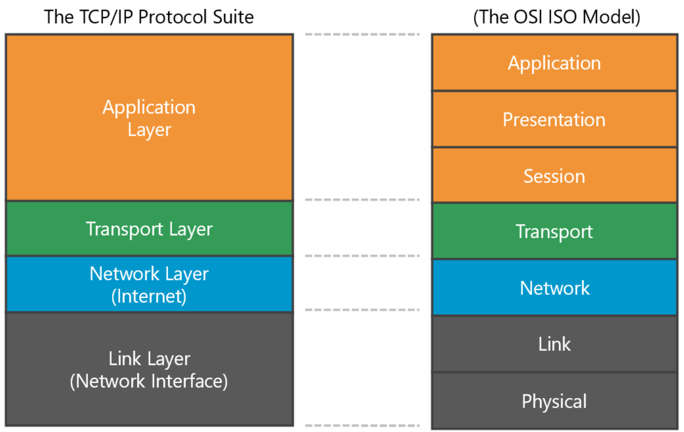
Layered Architecture

What are the benefits of having a layered architecture?

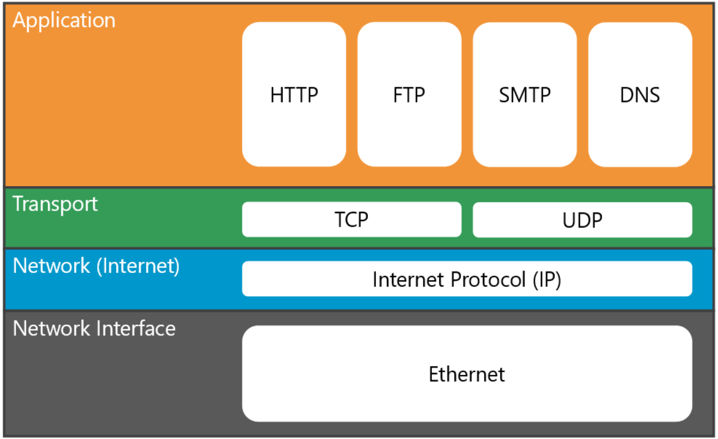
- It allows simplier design and implementation
- Each layer focus on different functions
- On each layer, protocols can be designed independently
- Different applications may choose to implement data communication on different layers depending on their requirements

Layered Architecture

The TCP/IP Protocol Suite

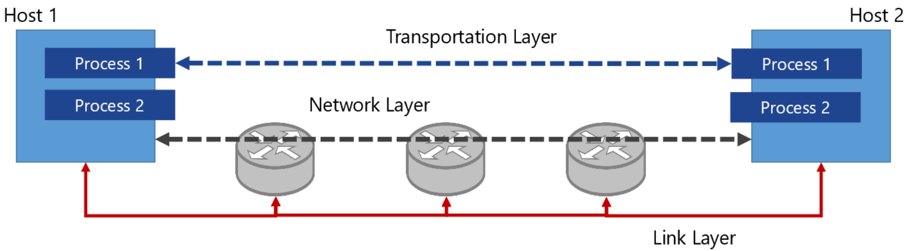


TCP/IP


Application
HTTP
FTP
SMTP
DNS
Transport
TCP
UDP
Network (Internet)
Internet Protocol (IP)
Network Interface
Ethernet

TCP/IP

- Transport Layer: responsible for process-to-process delivery
- Network Layer: host-to-host delivery
- Link Layer: node-to-node delivery (hop-by-hop)
- A process is an application program running on a host



Client/Server Model

- A process, called a **client**, requests services from a process on another host, called a **server**

- The following must be defined

    - Local host (Source IP address)
    - Local process (Source port number)
    - Remote host (Destination IP address)
    - Remote process (Destination port number)

- In client-server model, if we regard the client as the local host, then the server is the remote host, and vice versa

TCP/IP

There are three transport layer protocols defined in the TCP/IP Protocol Suite

- **User Datagram Protocol (UDP)**
- **Transmission Control Protocol (TCP)**
- Stream Control Transmission Protocol (SCTP)
  - New reliable and message-oriented protocol combines the best features of UDP and TCP
  - For streaming applications (e.g. video streaming)

TCP - The Transport Layer

Connectionless vs. Connection-oriented

- **Connectionless (UDP)**
  - No pre-established connection between sender and receiver
  - Packets are not numbered, and can arrive out of sequence
  - No acknowledgement of having received the packets
  - Unreliable
- **Connection-oriented (TCP)**
  - A connection is first established between the sender and the receiver
  - Has transport layer-level flow and error control
  - Reliable

vs.

UDP - User Datagram Potocol

Characteristics of UDP

- UDP is **connectionless** and **unreliable**
- Very simple using a minimum of overhead
- **Faster** and more efficient for many lightweight or time-sensitive purposes
- Suitable for processes sending small messages and does not care much about reliability
- Used for multicast and broadcast
- Common network applications that use UDP:
  - Domain Name System (DNS)
  - Trivial File Transfer Protocol (TFTP)

Datagram

UDP packets, called user datagrams, have a fixed-size 8 bytes header, containing 4 fields

1. **Source port number**
   - The port number used by the process running on the source host (16-bit)
2. **Destination port number**
   - The port number used by the process running on the destination host (16-bit)
3. **Length**
   - 16-bit field that defines the total length of the user datagram, header plus data (actually duplicated with the length field in IP)
   - UDP length = IP length – IP header's length
4. **Checksum**
   - A checksum for the user datagram

- *Question: how about IP address?*

TCP - Transmission Control Protocol

TCP: a stream-oriented protocol

- Instead of independent datagrams, TCP delivers data as **a stream of bytes**
- A large chunk of data is divided into **segments**, these segments are related to one another
- TCP creates an environment in which the two processes seem to be connected by an **imaginary tunnel**

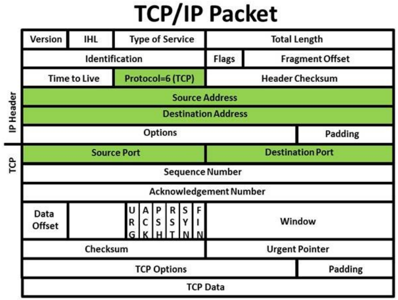
TCP

Flow Control

- The sending and the receiving processes may not write or read data at the same speed
- TCP needs **buffers for storage**, **flow control**, and **error control**
- One way to implement the buffer is to use a circular array of 1-byte locations
- TCP buffer size is configurable (e.g. buffer size = 2 *bandwidth* delay) (Can be up to megabytes)
- UDP does not have buffers and its queue length is relatively smaller

TCP

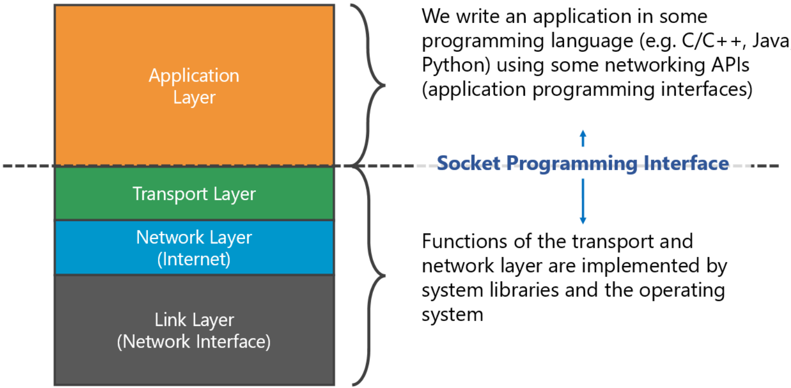
- TCP delivers data as **segments**
- TCP adds a header to each segment (for control purpose) and delivers the segment to the underlying IP layer for transmission
- The segments are encapsulated in IP datagrams and transmitted (The entire operation is transparent to the processes)



Network Programming and Socket Programming

Network Programming

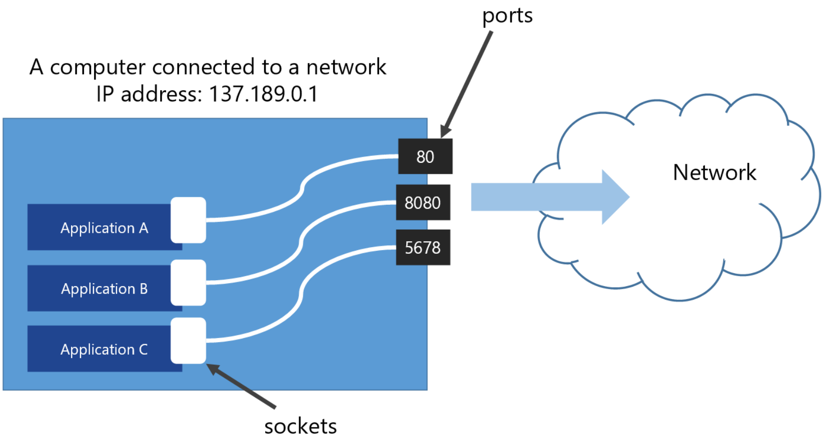
- What do we do in network programming?



Ports

- Ports are **"endpoints of communications"** in a computer's OS
- Ports allow different applications running on the same computer to share a single physical link to the network
- Each application must bind to a **unique port** (identified by a number) in order to communicate with the network

Ports


ports
A computer connected to a network
IP address: 137.189.0.1
80
8080
5678
Application A
Application B
Application C
Network
sockets

Ports

- Port number is a **16-bit unsigned integer** (i.e. 0 to 65535)
- Port numbers are regulated and are divided into **3 different ranges** (Regulated by the Internet Assigned Numbers Authority (IANA))
    - **Well Know Ports**
        - (0-1023) Registered for well-known applications
        - Example: 21 (FTP), 80 (HTTP), 443 (HTTPS), 465 (SMTPS)
    - **Registered Ports**
        - (1024-49151)
        - Registered for other applications
    - **Dynamic/Private Ports**
        - (49151-65535)
        - Can be used by private applications
    - Ref: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

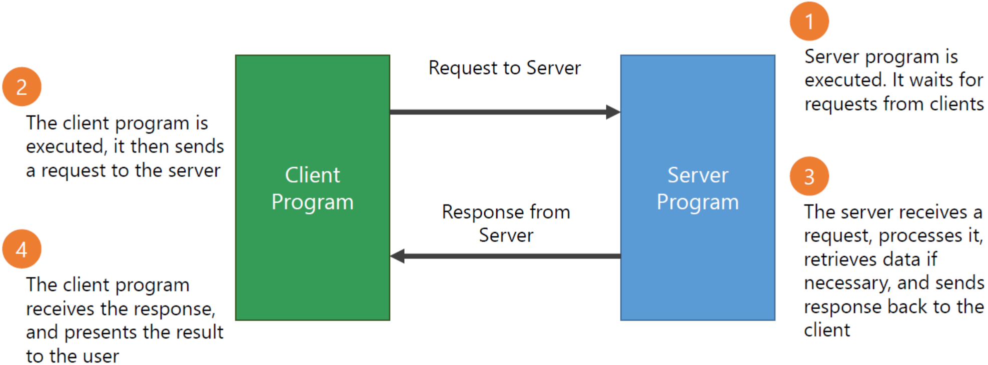
Using netstat

- You can use **`netstat`** to check the active network connections on your computer

```
$ netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:microsoft-ds          *:*                     LISTEN
tcp        0      0 localhost:8002          *:*                     LISTEN
tcp        0      0 *:8005                  *:*                     LISTEN
tcp        0      0 *:25672                 *:*                     LISTEN
tcp        0      0 *:8010                  *:*                     LISTEN
tcp        0      0 localhost:mysql         *:*                     LISTEN
tcp        0      0 localhost:6379          *:*                     LISTEN
tcp        0      0 *:netbios-ssn           *:*                     LISTEN
tcp        0      0 *:http                  *:*                     LISTEN
tcp        0      0 *:epmd                  *:*                     LISTEN
tcp        0      0 localhost:8500          *:*                     LISTEN
tcp        0      0 *:ssh                   *:*                     LISTEN
```

The Client-Server Model

- Many network applications follow the client-server model
- In such a model, servers are continuously running to wait for the request from clients



The Client-Server Model

Note:

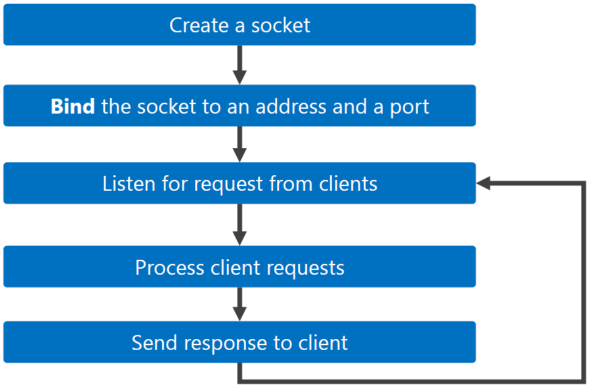
- “Client” and “server” here refer to the role of the program at some instance
- One application can be running both a client and a server at the same time
- A mobile app can be a server, if it is serving data to another mobile app

Questions:

- Can you think of examples of **servers** and **clients**?
- What would a server program do? What would a client program do?

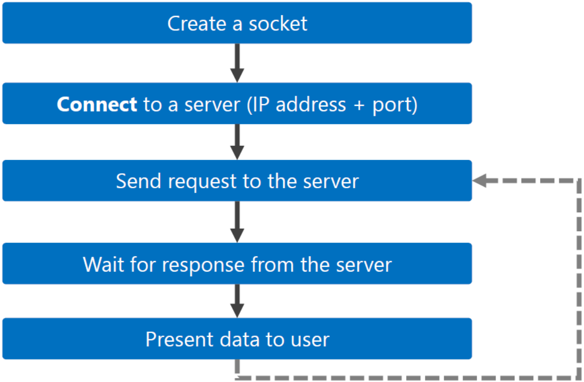
A Server Program

What does a server program do?



A Client Program

What does a client program do?



Socket Programming in Python

Sockets

- Used to identify a program (process) on a computer (host)
- A socket is defined by two numbers:
    1. **IP Address** (identifies the host)
    2. **Port number** (identifies the process)
- The commonly used implementation is called Berkeley Sockets or POSIX sockets
- Whenever a process on a host needs to talk to another process on another host, sockets must be created

- **Sockets in Python** is an object-based interface to the low-level operating system calls that are normally used to accomplish networking tasks on POSIX-compliant operating systems
- It exposes the normal **POSIX calls** for raw UDP and TCP connections

Server

The Steps to create a server program

1. **Create** a socket object in your program
2. **Bind** the socket object to a socket in the computer
3. **Listen** for incoming connection from clients
4. **Loop**:
    1. **Accept** connection from a client
    2. **Do** some stuff
    3. **Send** response to client (if necessary)
    4. **Close** the connection

Using Sockets in Python

- **NOTE**: Read the doc here: https://docs.python.org/3/library/socket.html

```python
import socket

# create an INET socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# bind the socket to the host and a port
server_socket.bind((socket.gethostname(), 50001))

# Listen for incoming connections from clients
server_socket.listen(10)
```

- **socket** is a module in the standard library in Python that provides access to the socket interface
- `socket.socket()` returns a new socket
- `socket.AF_INET` specifies that the socket will use the IPv4 family of address
- The second argument can either be `socket.SOCK_STREAM` (TCP) or `socket.SOCK_DGRAM` (UDP)

Using Sockets in Python

```
import socket

# create an INET socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# bind the socket to the host and a port
server_socket.bind((socket.gethostname(), 50001))

# Listen for incoming connections from clients
server_socket.listen(10)
```

- `server_socket` is an instance (object) of the socket class
- `socket.gethostname()` will return the name of the machine (try it yourself), this tells the socket to bind to interface that is visible to the public
- Instead, you can provide `"localhost"`, `"127.0.0.1"`, or even an empty string `""`
- `50001` is the port number that you want the socket to bind to

Using Sockets in Python

```
import socket

# create an INET socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# bind the socket to the host and a port
server_socket.bind((socket.gethostname(), 50001))

# Listen for incoming connections from clients
server_socket.listen(10)
```

- **NOTE**: `bind()` accepts only one argument, which is a tuple `(host_name, port_number)`
- Ask the socket to listen for incoming connection by using `listen()`
- `listen(10)` means that there can be at most 10 clients waiting in the queue (10 is the size of the **backlog**)

A TCP Server

Remember the steps?

1. ~~**Create** a socket object in your program~~
2. ~~**Bind** the socket object to a socket in the computer~~
3. ~~**Listen** for incoming connection from clients~~
4. **Loop**:
    1. **Accept** connection from a client
    2. **Do** some stuff
    3. **Send** response to client (if necessary)
    4. **Close** the connection

A TCP Server

```
# ...

# A indefinite loop
while True:
    # accept connections from outside
    (client_socket, address) = server_socket.accept()

    # Read data from client and send it back
    data = client_socket.recv(1024)
    client_socket.sendall(data)

    # Close the socket
    client_socket.close()
```

- `accept()` is a **blocking** call (wait until a client tries to connect)
- `recv(1024)` attempts to read 1,024 bytes from the client (also blocking)
- `sendall(data)` attempts to send all bytes of `data` to the client

Sending and Receiving Data

Behavour of `send()`

- We used `sendall()` in the above example to send data, which waits until ALL data has been sent out
- `send()` is a more generic function, which may or may not send out all data at once

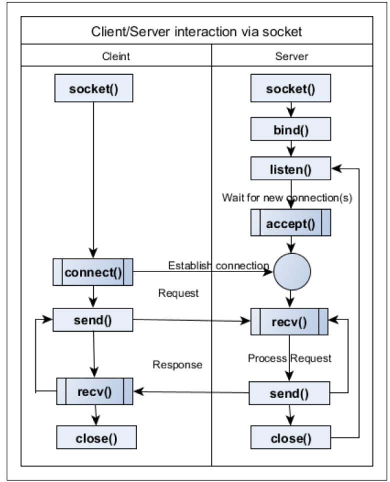
```
>>> len(data)
50000
>>> client_socket.send(data)
34582
...
```

- NOTE: TCP is a streaming protocol (remember `SOCKET.SOCK_STREAM`?), it does not have the concept of messages or records

A TCP Server

Notes:

- During **`accept()`**, a **new** socket is created and used solely for communication with that particular client
- Even though the client may send all data at the same time, **`recv()`** may not receive the data altogether
- **`recv()`** is a blocking function here
  - It will not return until some data is received in the buffer
  - If it returns an empty string, it means that the client has **disconnected** (the connection is broken)



A TCP Client

```
import socket

# create an INET TCP socket
soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# connect to the server (change localhost to an IP address if necessary)
soc.connect(("localhost", 50001))

# Send a message to the server
soc.send("Hello Server!".encode("utf-8"))

# Receive data from the server
data = soc.recv(1024)
print(data.decode("utf-8"))

# Always close the socket after use
soc.close()
```

Sending and Receiving Data

Behaviour of `recv()`

- It **blocks** until some data has arrived, and then it returns the data
- Even though the client sends everything in one go, the data might NOT be received at the server in all in one block (due to the nature of underlying IP network)
- If it returns with an empty string, it means that the connection has **dropped**

Question

- How do you know when to **stop** receiving?

A Protocol on Top of TCP

Design your own protocol to exchange data between server and client

1. **Fixed length message**
    - A message is always of the same length (e.g. 1024 characters)
    - Call `recv` until you get the whole message
2. **Delimit your message**
    - Add a special string (e.g. `"##THE END##"`) at the end of a message
    - Call `recv` until you see that special string
3. **Indicate message length in a header**
    - Include a fixed length header at the beginning of the message
    - Call `recv` to get the header to determine message lenght, and then call `recv` until message is received

- What are the **pros** and **cons**?

An Example of Fixed Lenght Message

```python
# Fixed length of a message
MSG_LENGTH = 2048

# Use a list to hold parts of the message received
parts = []
bytes_received = 0

while bytes_received < MSG_LENGTH:
    part = soc.recv(min(MSG_LENGTH - bytes_received, 1024))
    if part == b'':
        raise Exception("Connection is lost")
    parts.append(part)
    bytes_received += len(part)

# Join the parts into a single message
message = b"".join(parts)
```

UDP Server

```
import socket

# Note, the second parameter for socket() is socket.SOCK_DGRAM)
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind it to an IP address and a port
server_socket.bind((socket.gethostname(), 50001))

while True:
    # data is the data sent from a client
    # address is the IP address of the client
    data, address = server_socket.recvfrom(1024)
```

- we no longer need to call `listen()`
- we no longer need to call `accept()` (no connection will be established)

UDP Client

```
import socket

client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Server address and port number, and message to be sent
server_address = ("localhost", 50001)
message = "I love socket programming in Python!"

bytes_send = client_socket.sendto(message, server_address)

# Close the socket
socket.close()
```

More on Python Programming

Let's go through some examples to let you get familiar with programming in Python

Python Programming Example 1

Given a string, count the occurrences of each character and return the result in a dictionary

```
sentence = "Python programming is fun!"

# Declare an empty dictionary
output = {}

# Loop through each character in the sentence
for char in sentence:
    # .get returns the value if "char" is a key in the dictionary
    # otherwise the second argument is return (0 in this case)
    output[char] = output.get(char, 0) + 1

print(output)
# prints {'n': 3, 't': 1, 'P': 1, 'o': 2, 's': 1, ...
```

Python Programming Example 2

Given a list, filter away some elements that do not satisfy some conditions

```
cities = ["Hong Kong", "Paris", "London", "Tokyo", "New York"]

# Create a new list with elements that contains the character "k"
with_k = [x for x in cities if "k" in x]

# Create a new list with elements whose length is larger than 5
longer_than_5 = [x for x in cities if len(x) > 5]

# Create a new list with elements that have two words
two_words = [x for x in cities if len(x.split(" ")) == 2]
```

Python Programming Example 3

Combining two lists in parallel

```
students = ["Chan", "Yip", "Leung", "Zhang", "Xu"]
grades = ["B", "A-", "B+", "C+", "A"]

# Use zip to generate a list of tuples from two lists
for student, grade in zip(students, grades):
    print("%s gets %s in this course" % (student, grade))
```

- `zip()` receives an arbitrary number of arguments (try providing 3 or even 4 lists!)

Python Programming Example 4

Generating an index when you are looping through a list

```python
cities = ["Hong Kong", "Paris", "London", "Seoul", "Singapore"]

# Naive method
i = 1
for c in cities:
    print("Number %d: %s" % (i, c))

# Better
for i, c in enumerate(cities, 1):
    print("Number %d: %s" % (i, c))
```

- **`enumerate()`** returns a list of tuples with the first one as the index
- The second argument of **`enumerate()`** allows you to specify the starting index

Python Programming Example 5

Using `try...except`

```
import json

# Malformatted JSON string
raw_data = '{"key1": "value1", "key2": "value2"'

try:
    data = json.loads(raw_data)
except Exception as err:
    print("Runtime error:", err)
```

Python Programming Example 6

Write your own class

```
# Define a class using the class keyword
class MyClass:

    # Every class has a constructor
    # Every function in a class receives the self argument
    def __init__(self):
        self.data = "default data"

    # Access attributes of the class using self.
    def print_data(self):
        print(self.data)


# Create a new instance of the class
x = MyClass()
x.print_data()
```

Python Programming Example 7

Passing arguments through the command line

```
# Suppose you invoke your program like this
$ python3 my_script.py data1 5678


# In my_script.py, you can access the arguments as follows
import sys

num_arguments = len(sys.argv)  # this returns 3

arg1 = sys.argv[1]  # arg1 now contains "data1"
arg2 = sys.argv[2]  # arg2 now contains "5678"
```

End of Lecture 2