**1. 网络编程：**

应用调用操作系统，操作系统控制硬件

osi七层协议：应用层，表示层，会话层，传输层，网络层，数据链路层，物理层

传统的五层协议：应用层（应用层，表示层，会话层），传输层，网络层，数据链路层，物理层。

**客户端：**

应用层： 应用产生数据，遵循协议（如:http,ftp协议）的标准来对数据包装

传输层： 通过TCP/UDP端口协议传输（端口范围0—6535）

网络层： 遵循IP协议；由于客户端和服务端可能在不同的子网中，要想发送接收方就需要找到目标的具体位置。

数据链路层： 遵循以太网协议；其中[报头：data]；报头18个字节，前6位源地址后六位目标地址，然后6位描述信息。网卡（eth0）。用MAC地址通过广播形式找到**同一子网（局域网）**目标地址位置，通过MAC+IP可以找到全世界目标在网中的具体位置

物理层： 比如数据链路层的数据解析为01011001通过网络并传输。

应用层 客户端产生数据，通过协议包装。服务端协议解包装，获取数据

传输层 通过tcp/ip发送数据，即要知道发给哪个端口

网络层 包装 ip头：数据 IP

数据链路层 在包装 报头：ip头：数据 IP+MAC(arp协议解析ip对应的mac)

物理层 解析为二进制并通过网络传输

**TCP/IP协议（数据流形式发送）：**

TCP：三次握手和四次挥手。效率低，可靠性高

三次握手：

1. 客向服发起请求连接syn=1。

2.1 服确认客的请求ack=1。

2.2 服向客连接syn = 1。

3. 客收到服的确认信息。

2.1和2.2可以合成一步，即客同意服的请求，并建立连接

四次挥手

1. 客向服发起断开请求FIN=1,此时客不在向fuw发送数据。

2.服确认客的断开请求ack=1，此时服可能还没发完客请求的数据

3. 服向客发送数据已经完毕，准备断开FIN=1

4. 客收到服的信息，并确认ack=1。

2 和 3 不 可以合成一步，因此为服可能还没发完客请求的数据

UDP：效率高，可靠性低

**socket：**

socket是基于C/S架构的，也就是说进行socket网络编程，通常需要编写两个py文件，一个服务端，一个客户端。

服务端（tcp）：

import socket

# 绑定端口端口范围0——6535,前1024为给操作系统用

ip\_port = ('127.0.0.1', 9999)

# 默认使socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)，即tcp协议

sk = socket.socket() # 创建套接字

sk.bind(ip\_port) # 绑定服务地址

sk.listen(5) # 监听连接请求

print('启动socket服务，等待客户端连接...')

conn, address = sk.accept() # 等待连接，此处自动阻塞

while True: # 一个死循环，直到客户端发送‘exit’的信号，才关闭连接

client\_data = conn.recv(1024).decode() # 接收信息 1024表示最大可接受字节数
if client\_data == "exit": # 判断是否退出连接

exit("通信结束")

print("来自%s的客户端向你发来信息：%s" % (address, client\_data))

conn.sendall('服务器已经收到你的信息'.encode()) # 回馈信息给客户端

conn.close() # 关闭连接

sk.close()

客户端（tcp)：

import socket

ip\_port = ('127.0.0.1', 9999)

s = socket.socket() # 创建套接字

s.connect(ip\_port) # 连接服务器

while True: # 通过一个死循环不断接收用户输入，并发送给服务器

inp = input("请输入要发送的信息： ").strip()

if not inp: # 防止输入空信息，导致异常退出

continue

s.sendall(inp.encode())

if inp == "exit": # 如果输入的是‘exit’，表示断开连接

print("结束通信！")

break

server\_reply = s.recv(1024).decode()

print(server\_reply)

s.close() # 关闭连接

**想要执行系统命令需要调用import os或者 import subprocess**

1. import os

os.system('ls /')

# 如果为linux系统，则执行ls命令，但是默认输出到终端，不能保存

2. import subprocess

obj = subprocess.Popen('ls /', shell=True,

stdout=subprocess.PIPE,

stderr=subprocess.PIPE,)

# windows默认gbk编码

# shell=True,表示可以以shell方式运行命令，

# stdout,stderr是输出管道，如果命令错误则输出stderr里管道的内容

print('stdout--->1', obj.stdout.read().decode('utf-8'))

print('stderr--->1', obj.stderr.read().decode('utf-8'))

**如下是windows**

a = subprocess.Popen('dir c:\\', shell=True,

stdout=subprocess.PIPE,

stderr=subprocess.PIPE,)

print(a.stdout.read().decode('gbk'))

**粘包：**

**理解：**

**1. 不管recv还是send都不是直接接受对方资源，而是操作自己的操作系统内存**

**即不是一个send对应一个recv**

**2. send方法时间特点：**速度快，（应用层）只需要把发送的数据复制给操作系统内存，剩下的全部交给操作系统处理（把应用上的数据通过操作系统读取到硬盘中 ）

**3. recv方法时间特点：**

1. 等待时间长

2. copy时间（把硬盘上的数据通过操作系统读取到应用中 ）

**原因：**

发送端为了将多个发往接收端的包更有效的发到对方，使用了优化算法（Nagle）：

即将多次间隔较小且数据量较小的数据包合并成一个数据块，然后进行封包。

**即粘包是由于Nagle算法特性决定的**

**2. 并发编程**