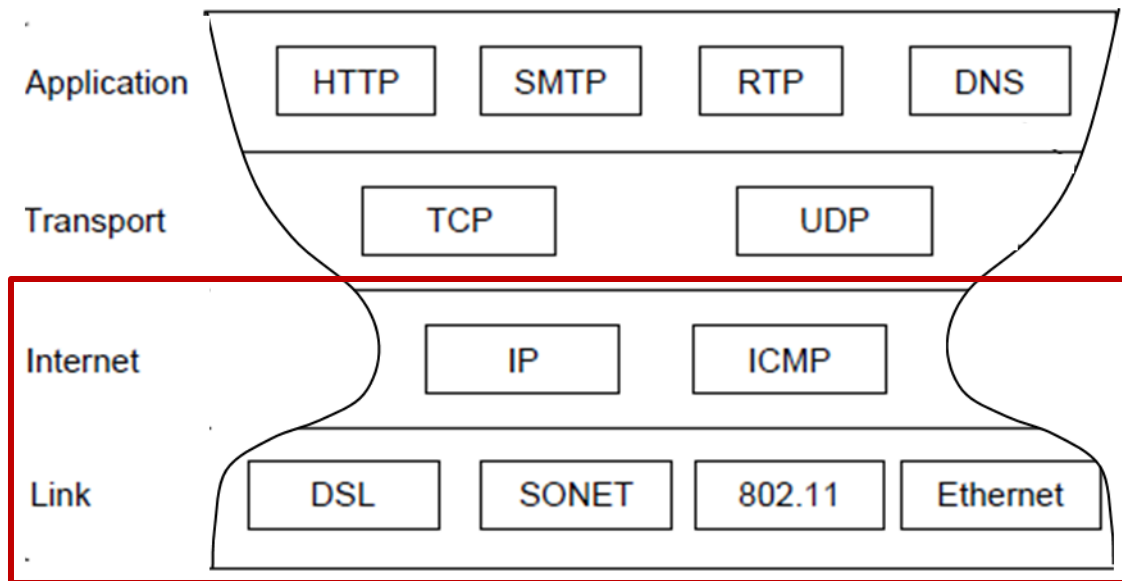


CSE 3231
Computer Networks
Socket Programming

William Allen, PhD
Spring 2022

The TCP/IP Model

The four layers of the **TCP/IP** reference model provide modularity, flexibility and reliability to support many different network applications

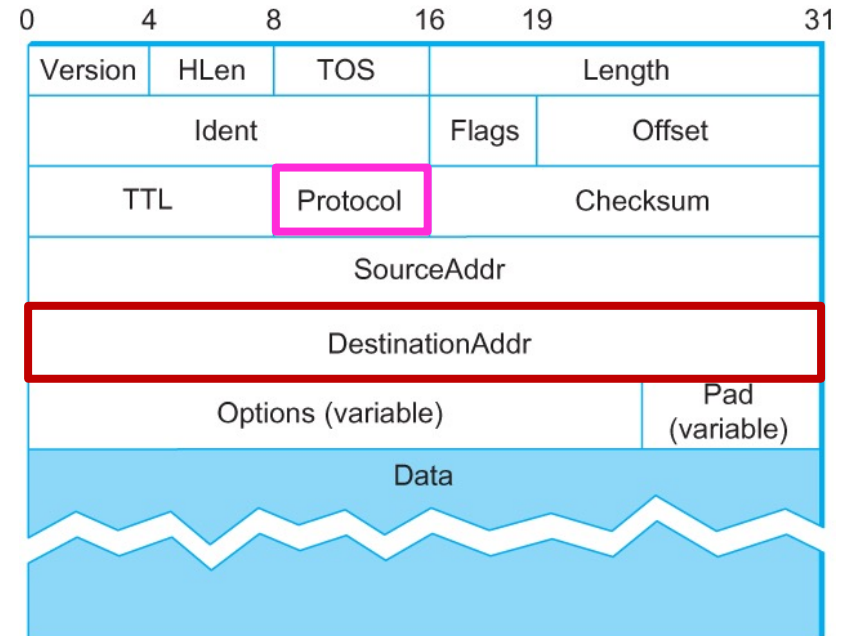


Protocols are shown in their respective layers

So far, we have looked at the lower two layers of the TCP/IP model used in the Internet

IP Packet Format

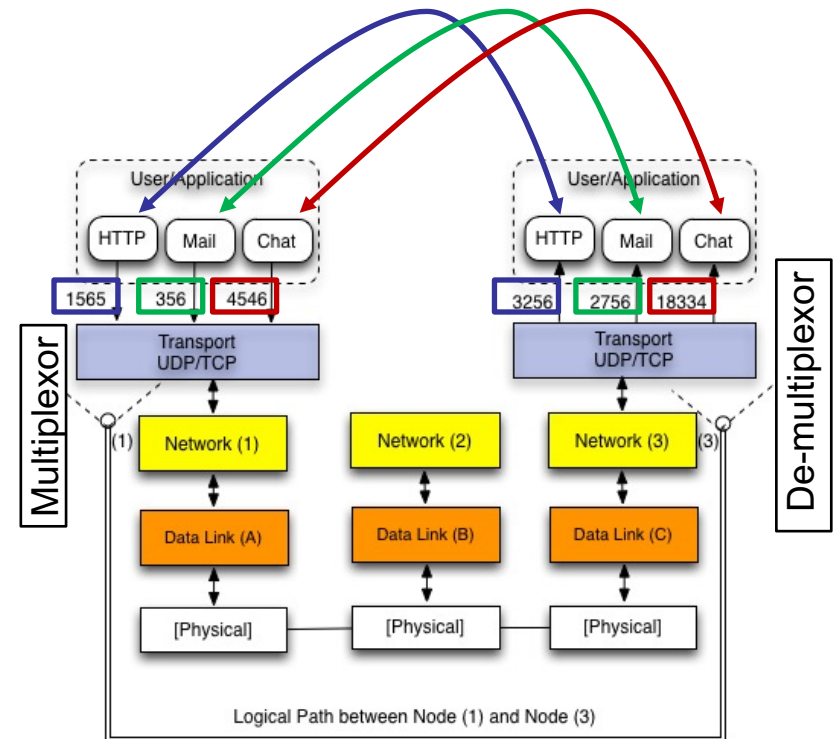
- **Version** (4 bits): most often v4
- **Hlen** (4 bits): size of the header
- **TOS** (8 bits): type of service
- **Length** (16 bits): size of packet
- **Ident** (16 bits) and **Flags/Offset** (16 bits): supports fragmentation
- **TTL** (8 bits): number of hops
- **Protocol** (8 bits): demultiplex key
- **Checksum** (16 bits): (header only)
- **SrcAddr**, **DestAddr** (32 bits each)



- The two fields that we need to focus on when writing code for networking are the *protocol* and *destination address*.
- The rest of the header fields are filled in by the network protocol software, often with default values.

Network Applications and Ports

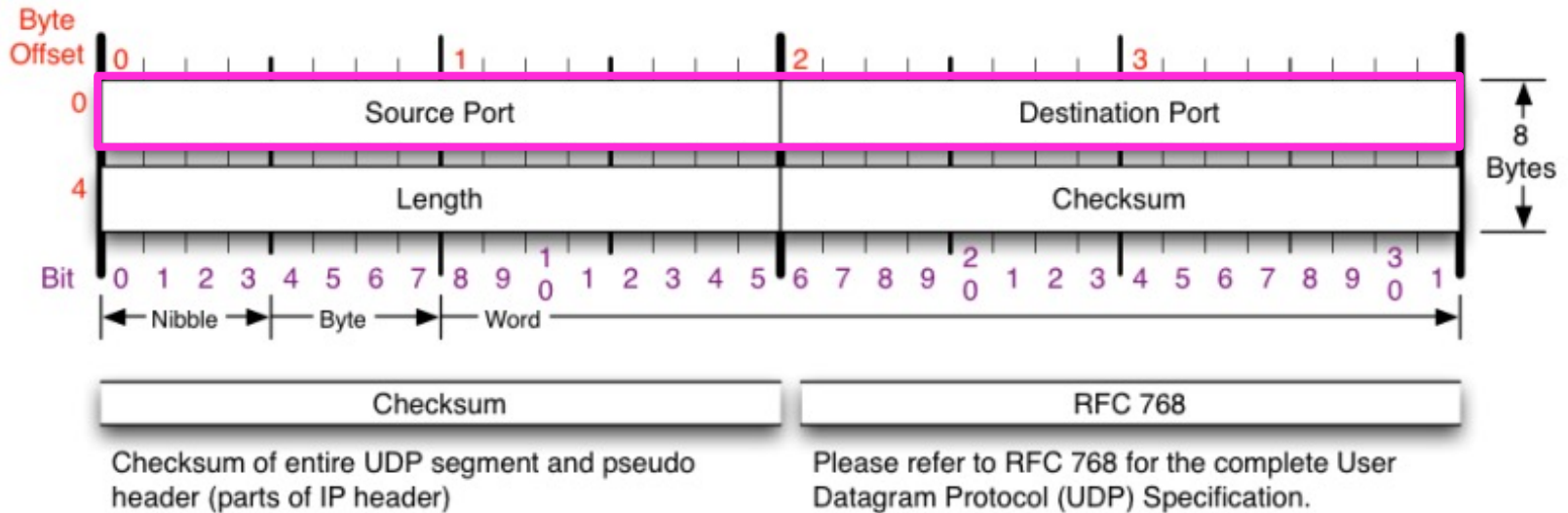
- Most computer users are interacting with multiple applications concurrently
- While the destination host can be identified by its IP address, it may be providing several different network services and the IP address isn't enough to identify which one is in use
- Both UDP and TCP use *ports* to *multiplex* the segments at the sender side and *demultiplex* the segments at the receiver side



Preview of TCP and UDP

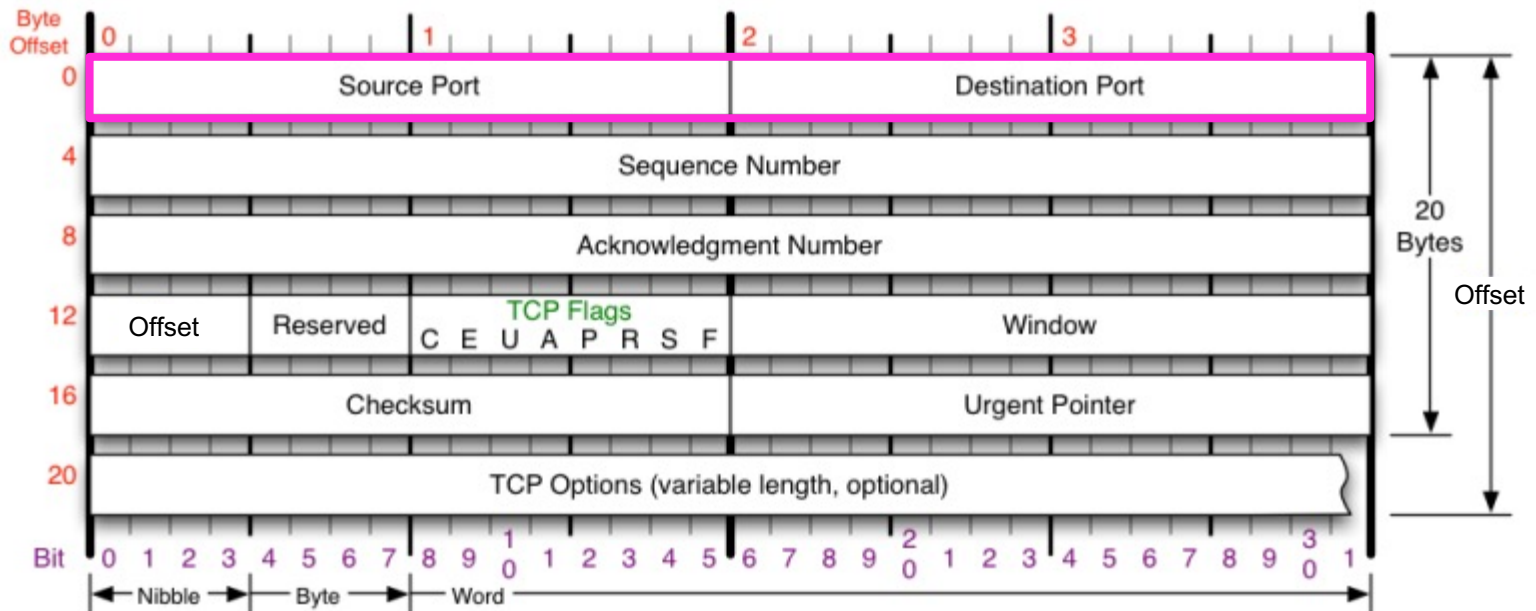
- The **Network layer** sends and receives packets between specified **IP** addresses and can handle congestion by lowering transmission rate
- The **User Datagram Protocol** (UDP) in the **Transport layer** can direct packets to specific applications on the destination node
- The **Transport Control Protocol** (TCP) can establish end-to-end connections and manage the delivery of packets across the network
 - TCP headers include information to manage connections and adapt to changing conditions

UDP Header



- The header for UDP (User Datagram Protocol) has only four fields and we are normally only concerned with the *port numbers*
- The source port number is often chosen by the network software from a pool of available ports, we only choose the destination port, which depends on the application

TCP Header



- The header for TCP (Transport Control Protocol) has many more fields, but we are still most often focused on the *port numbers* when writing networking software
- As with UDP, the source port number is often chosen by the network software from a pool of available ports and we only choose the destination port

Berkeley Sockets

Very widely used primitives developed at UC Berkeley to support TCP networking on UNIX

- “*sockets*” are used as transport layer endpoints
- Originally accessed in C/C++, now available in all major programming languages

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Berkeley Sockets

Some primitives are similar to those on the earlier list, but others were developed to better control creation of transport-level connections

- **SOCKET** and **BIND** are used to designate the connection endpoint on a server
 - clients also use **SOCKET**, but not **BIND**
- **LISTEN** and **ACCEPT** are used by *servers* to wait for a connection request (not used by clients)
- **CONNECT**, **SEND** and **RECEIVE** work as described before
- **CLOSE** ends a connection (similar to closing a file)

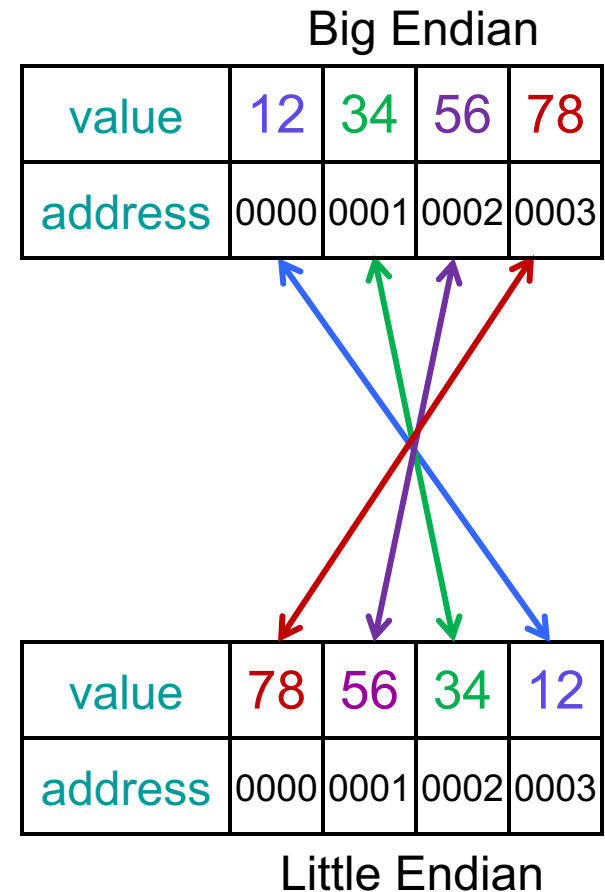
Big Endian vs Little Endian

Big Endian byte order

- stores multi-byte values with the byte representing the highest powers of 2 at the left (or 'big') end
- for example: 0x**1****2****3****4****5****6****7****8**

Little Endian byte order

- stores multi-byte values with the *byte* representing the highest powers of 2 at the right (or 'little') end



Data Formats and Byte Order

- Networks use **Big-Endian** format for multi-byte data and many CPU's (e.g., Intel) store data in **Little-Endian** format
 - C and Python provide functions to convert byte order
 - **htons()** (**h**ost to **net** **s**hort) converts **short int** (16-bit)
 - there is also a **htonl()** to convert 32-bit integers and **ntohs()** (16-bit) and **ntohl()** (32-bit) to convert data from Big-Endian (net) to Little-Endian (host)
 - Java stores data in Big-Endian format, no conversion
- ASCII text is transmitted as individual bytes and does not have to be converted

Data Formats and Byte Order

- Dotted IP addresses are stored in a string and 32-bit binary addresses in an `in_addr` struct

```
struct in_addr{  
    in_addr_t s_addr; /*32 bit IPv4 network address*/  
};
```

- Since network IP addresses are often shown in 'dotted' notation (132.168.1.254) there are also functions to convert them into and out of binary

`inet_ntoa(in_addr n)` converts numbers to strings

`inet_aton(char * ip)` converts strings to numbers

– `ntoa` == net to ASCII, `aton` == ASCII to net

Blocking and Non-Blocking I/O

- The term “*blocking*” refers to whether the network I/O function waits for the other end of the connection
 - *recv()* functions wait for the sender to send data and, therefore, are blocking functions
 - *accept()*, *connect()*, *send()* usually also block, but non-blocking versions are possible too
 - however, then it is up to the programmer to determine what to do next
 - this can lead to *polling* where the program has to keep checking to see if the call was successful

Types of Socket

- Servers may use one socket to receive new connections (a *listening* or *server* socket) and another for the exchange of data
- When the connection is accepted, the *listening socket* hands over the connection to the second socket, allowing it to return to listening for new connection requests

```
ServerSocket sock = new ServerSocket(portnum);  
Socket clientSocket = sock.accept();
```

- The *ServerSocket* listened for a connection and then the new *clientSocket* was created to exchange data after the connection was accepted

Options and Data Structures

- **C** provides constants for configuring sockets:
 - For IPv4 connections: `AF_INET`, for IPv6: `AF_INET6`
 - TCP connections: `SOCK_STREAM`
 - UDP packets: `SOCK_DGRAM`
 - Client IP addresses that the server will accept can be specified or connections can come from any host:
source address = `INADDR_ANY`
- There are also constants for socket options:
 - `SO_KEEPALIVE` - connections are kept alive
 - `SO_REUSEADDR` - reuse of local addresses is supported
 - `SO_BROADCAST` - can transmit broadcast messages
 - and many others

C - TCP Server

```
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#define PORT 12345
```

Note: in C, file descriptors are used for network connections because UNIX systems consider reading and writing over the network to be just like file I/O

```
int main(int argc, char const *argv[]) {
    int serv_fd, new_sock;
    struct sockaddr_in addr;
    int opt = 1;
```

Creating a file descriptor for an IPv4 TCP socket

```
serv_fd = socket(AF_INET, SOCK_STREAM, 0);
```

Setting socket options

```
setsockopt(serv_fd, SOL_SOCKET, SO_REUSEADDR|SO_REUSEPORT, &opt, sizeof(opt));
```

```
addr.sin_family = AF_INET;
```

```
addr.sin_addr.s_addr = INADDR_ANY;
```

```
addr.sin_port = htons( PORT );
```

Accept any source address, but use the designated port number

```
bind(serv_fd, (struct sockaddr *) &addr, sizeof(addr));
```

Binding the socket to that address

```
listen(serv_fd, 3);
```

```
new_sock = accept(serv_fd, (struct sockaddr *) &addr, (socklen_t*)&addrlen);
```

```
}
```

Listening for a new connection and accepting it when the client connects

C - TCP Client

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#define PORT 12345
```

```
int main(int argc, char const *argv[]) {
    int sock = 0;
    struct sockaddr_in serv_addr;
```

```
    sock = socket(AF_INET, SOCK_STREAM, 0);
```

```
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
```

Configuring the socket for a connection using IPv4 and TCP

```
// Convert the IPv4 address from text to binary form
    inet_pton(AF_INET, "192.168.1.88", &serv_addr.sin_addr);
```

```
    connect(sock, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
```

Interaction with the server goes here.

```
}
```

C - UDP Client

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#define PORT 12345
```

```
int main(int argc, char const *argv[]) {
    int sock = 0;
    struct sockaddr_in serv_addr;
```

```
    sock = socket(AF_INET, SOCK_DGRAM, 0);
```

```
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
```

Configuring the socket for a connection using IPv4 and UDP

```
// Convert the IPv4 address from text to binary form
    inet_pton(AF_INET, "192.168.1.88", &serv_addr.sin_addr);
```

```
    connect(sock, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
```

Interaction with the server goes here.

```
}
```

Java – TCP Server

```
import java.io.*; import java.net.*; ◀... import io, network libraries
```

```
class simpleServer {  
    public static void main (String args[]) throws IOException {  
        int portnum = 12345;  
        ServerSocket sock = null; ◀... variable for server's socket,  
        try { ◀... bound to port when created  
            sock = new ServerSocket(portnum); // connect to port #  
        }  
        catch (IOException e) {  
            System.out.println("Could not listen on port: " + portnum + e);  
            System.exit(1);  
        }  
        Socket clientSocket = null; ◀... variable for client socket when connected  
        ◀... (note that it is a different type of socket)  
        try { ◀... listen for connection from client  
            clientSocket = sock.accept(); // listen for connection  
        }  
        catch (IOException e) {  
            System.out.println("Accept failed: " + portnum + ", " + e);  
            System.exit(1);  
        }  
    }  
}
```

try/catch
for error
handling

try/catch
for error
handling

Java - TCP Client

```
import java.io.*; import java.net.*; ◀... import io, network libraries
```

```
class simpleClient ◀..... Java class for this client program
```

```
public static void main (String args[]) throws IOException {
```

```
String host = "192.168.1.88";
```

```
int portnum = 12345;
```

designate server's IP and port

create new socket for client and assign to server's IP and port number

```
Socket sock = new Socket(host, portnum); // connect to server
```

note that this
is not a
ServerSocket

```
System.out.println("\nConnection established.");
```

Interaction with the server goes here.

```
}
```

```
}
```

Java - UDP Client

```
public static void main (String args[]) throws IOException {  
  
    byte[] buf = new byte[64];  
    int portnum = 12345;  
    String server = "192.168.1.88";  
    InetAddress serverAddr = InetAddress.getByName(server);  
  
    DatagramSocket sock = new DatagramSocket();  
  
    DatagramPacket packet =  
        new DatagramPacket(buf, buf.length, serverAddr, portnum);  
  
    sock.send(packet);  
  
    packet = new DatagramPacket(buf, buf.length);  
  
    sock.receive(packet);  
  
    sock.close();  
}
```

Datagram = UDP

Note: not a complete program

Python – TCP Server

```
import socket
PORT = 12345
soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
soc.bind(('', PORT))
soc.listen()
print('\nListening port ' + str(PORT) + '\n')
while True:
    conn, (client_host, client_port) = soc.accept()
    print('Connection from ', client_host, client_port)
```

import socket library

designate port to listen on for new connections

Port to listen on (privileged ports < 1024)

create new socket using IPv4 (AF_INET) and TCP (SOCK_STREAM)

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as soc:

bind the new socket to the port number

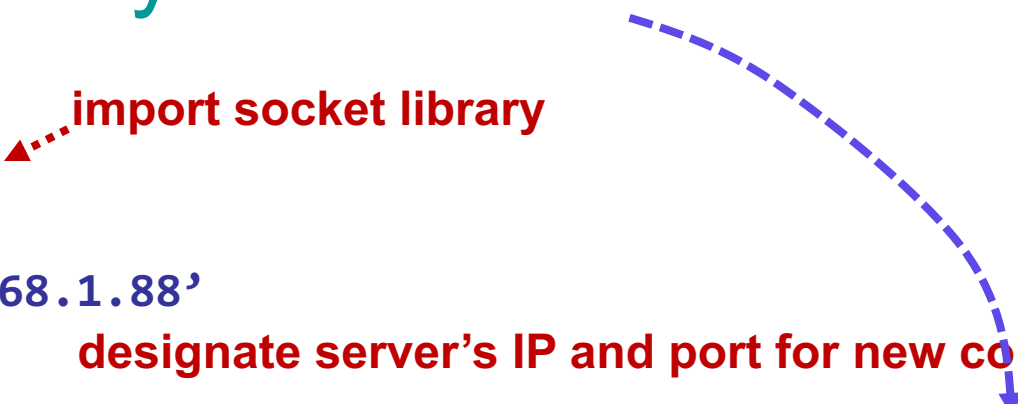
start listening for new connections

when a new connection is accepted, conn points to the connection, client_host = IP and client_port = port of connecting host

conn, (client_host, client_port) = soc.accept()

print('Connection from ', client_host, client_port)

Python – TCP Client



```
import socket
HOST = '192.168.1.88'
PORT = 12345
soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
soc.connect((HOST, PORT))
print('Connected to server: ', HOST, PORT)
```

import socket library

designate server's IP and port for new connections

create new socket using IPv4 (AF_INET) and TCP (SOCK_STREAM)

connect to the server using the specified port number

Interaction with the server goes here.

Note: not a complete program.

Python – UDP Client

import socket library
`import socket`

`HOST = '192.168.1.88' # IP address of server`

`PORT = 12345`

`serverAddr = (HOST, PORT)`

create new socket using IPv4 (AF_INET) and TCP (SOCK_DGRAM)

`UDPsoc = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)`

UDP – no connection

Interaction with another computer goes here.

Note: not a complete program.

Transferring Data

- C, Java and Python send and receive data much like file I/O
 - The syntax may be different, but all of them use some type of *send* and *receive* functions to transfer data
 - Text transfers may require encoding
- As with files, the connection should be closed when the transfer of data is done, but it will also automatically be closed when the program ends
- However, the socket binding may have to timeout before that address/port can be used again
 - The socket option `SO_REUSEADDR` can allow immediate reuse of local addresses

Transferring Data in C

- Since C treats network I/O the same as files, it has `read()` and `write()` functions for networks, but network I/O is more complex than file I/O
 - files are either accessible or not and you find out whether they are when you do a `file open()`
 - network connections may be slow, may be affected by congestion or dropped packets, or may disconnect
 - networks transfer data in “chunks” and parts of the data may not arrive in the order they were sent
 - thus, file-like `read()` and `write()` operations may not be flexible enough for the application you are coding
 - the following I/O functions support additional options

Transferring Data in C

- C functions for TCP connections:

```
int count = send(socket_fd, msg, len, flags);
```

`msg` is in a char buffer and `len` bytes will be sent

```
int count = recv(socket_fd, buf, len, flags);
```

`buf` is the receive buffer for no more than `len` bytes

- C functions for UDP datagrams:

```
int count = sendto(socket_fd, msg, len,  
flags, &serverAddr, addrlen))
```

```
int count = recvfrom(socket_fd, buf, len,  
flags, &clientAddr, addrlen);
```

- C has a close function for both packet types:

```
int status = close(socket_fd);
```

Transferring Data in Java

- **Java** has methods for sending and receiving streams of data and the following are often used for text input and output

- The `PrintWriter` class can write to a socket

```
PrintWriter out = new PrintWriter (socket.getOutputStream(), true);  
out.println("Hello Server");
```

- The `BufferedReader` class can read from a socket

```
BufferedReader input = new BufferedReader(new  
    InputStreamReader(socket.getInputStream()));  
String line = input.readLine();
```

Transferring Data in Python

- **Python** also has functions for sending and receiving streams of data

```
msg = conn.recv(1024)    # receives up to 1024 bytes
```

```
# Note: in Python 3, the received data is bytes, not a  
# text string (Unicode or other formats) and you  
# would have to decode() the string to get ASCII text
```

```
conn.sendall(msg)        # sends the data in msg  
# depending on the type of data sent and version of  
# Python, you may need to use the encode() function  
# to convert the data into bytes for transmission
```

Note: *sendall()* will send the whole buffer, *send()* may not

Python Server Program

```
#!/usr/bin/env python3
import socket
HOST = '192.168.1.88' # IP of this server
PORT = 12345          # Port to listen on

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as soc:
    soc.bind((HOST, PORT))
    soc.listen()
    print('\nListening on '+HOST+' using port '+str(PORT)+'\n')
    conn, addr = soc.accept()
    with conn:
        print('Connection from ', addr)
        while True:
            data = conn.recv(1024).decode()
            conn.sendall(('ECHO: '+data).encode())
            if (not data) or (data == 'quit'):
                break
        conn.close()
```

Python Client Program

```
#!/usr/bin/env python3
import socket
HOST = '192.168.1.88' # The server's hostname or IP address
PORT = 12345          # The port used by the server
line_in = ' '

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as soc:
    soc.connect((HOST, PORT))
    print('Connected to :', HOST, PORT)
    for i in range(3):
        line_in = input('\nEnter text: ')
        soc.sendall(line_in.encode())
        data = soc.recv(1024)
        print('Received: '+data.decode())
```