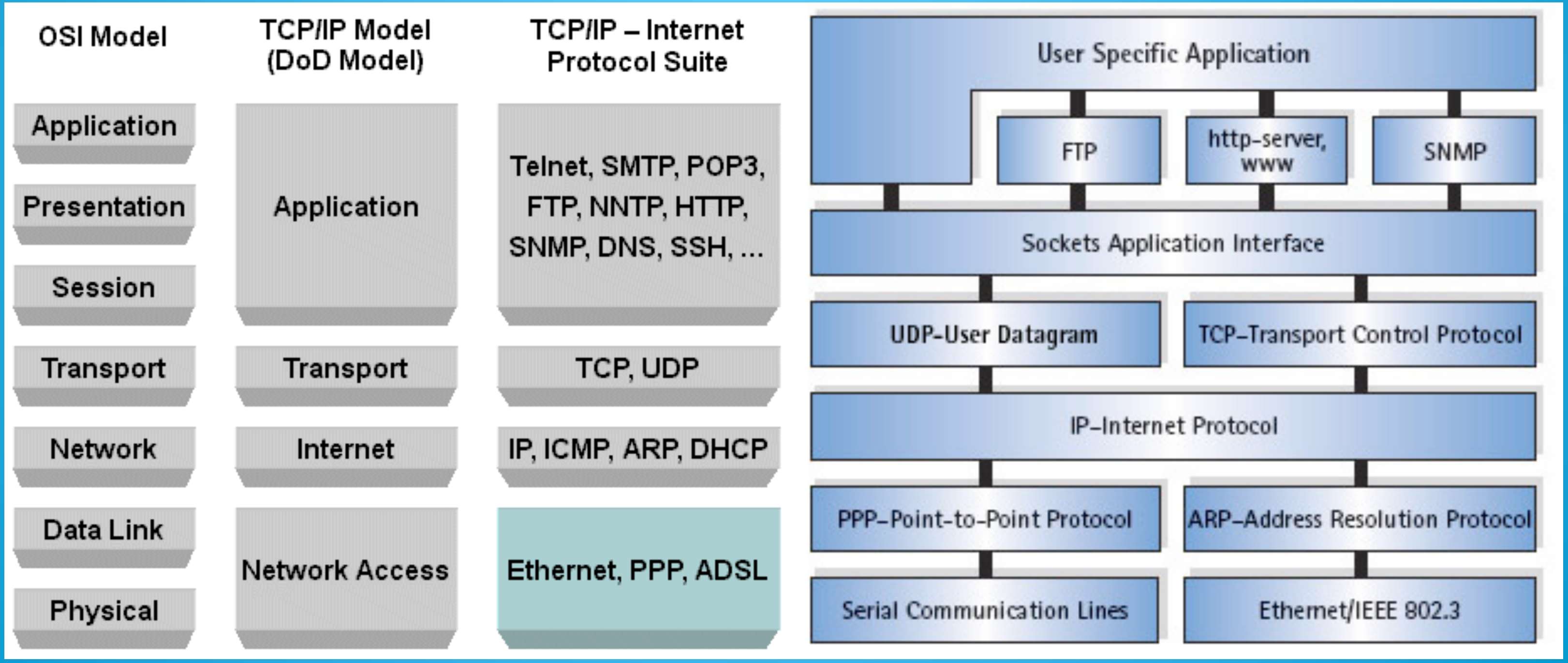
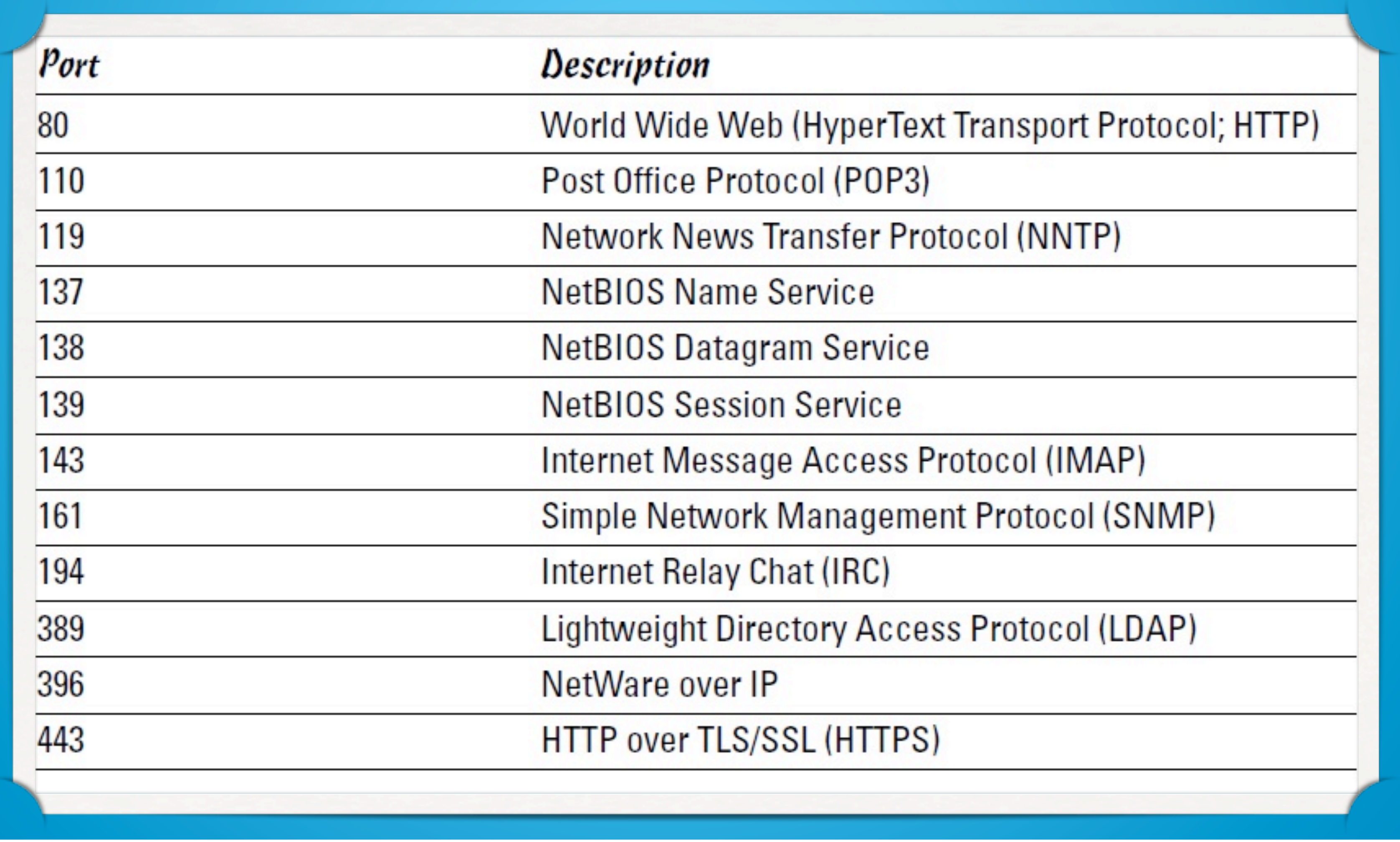
TCP



OSI Model	TCP/IP Model (DoD Model)	TCP/IP – Internet Protocol Suite
Application	Application	Telnet, SMTP, POP3, FTP, NNTP, HTTP, SNMP, DNS, SSH, ...
Presentation		
Session		
Transport	Transport	TCP, UDP
Network	Internet	IP, ICMP, ARP, DHCP
Data Link	Network Access	Ethernet, PPP, ADSL
Physical		

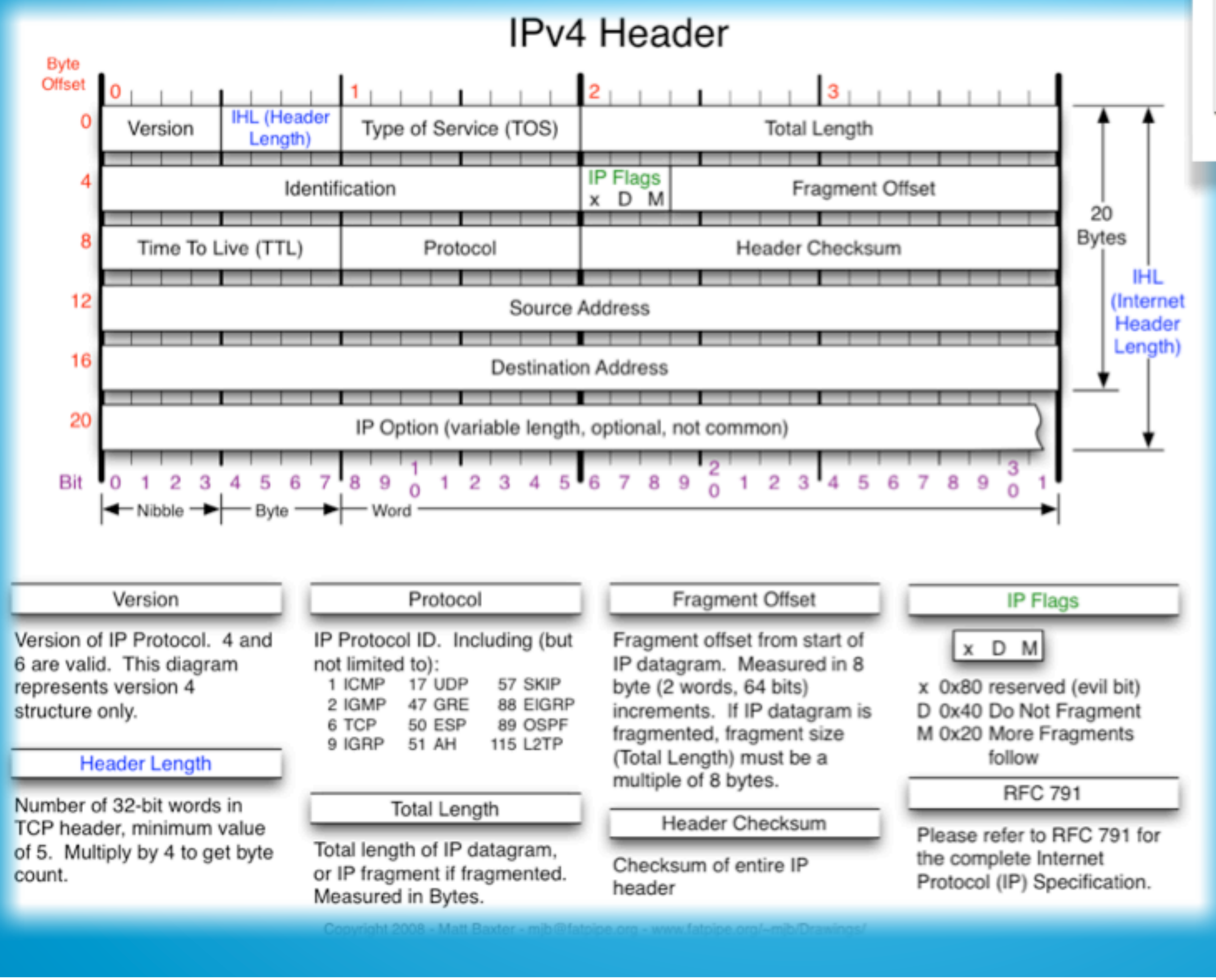
User Specific Application
FTP
http-server, www
SNMP
Sockets Application Interface
UDP–User Datagram
TCP–Transport Control Protocol
IP–Internet Protocol
PPP–Point-to-Point Protocol
ARP–Address Resolution Protocol
Serial Communication Lines
Ethernet/IEEE 802.3

TCP



Port	Description
80	World Wide Web (HyperText Transport Protocol; HTTP)
110	Post Office Protocol (POP3)
119	Network News Transfer Protocol (NNTP)
137	NetBIOS Name Service
138	NetBIOS Datagram Service
139	NetBIOS Session Service
143	Internet Message Access Protocol (IMAP)
161	Simple Network Management Protocol (SNMP)
194	Internet Relay Chat (IRC)
389	Lightweight Directory Access Protocol (LDAP)
396	NetWare over IP
443	HTTP over TLS/SSL (HTTPS)

TCP



IPv4 Header

Byte Offset	0		1	2		3
0	Version	IHL (Header Length)	Type of Service (TOS)	Total Length		
4	Identification			IP Flags x D M	Fragment Offset	
8	Time To Live (TTL)		Protocol	Header Checksum		
12	Source Address					
16	Destination Address					
20	IP Option (variable length, optional, not common)					

20 Bytes
IHL (Internet Header Length)
Bit 0 1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 20 1 2 3 4 5 6 7 8 9 30 1
Nibble
Byte
Word

Version

Version of IP Protocol. 4 and 6 are valid. This diagram represents version 4 structure only.

Header Length

Number of 32-bit words in TCP header, minimum value of 5. Multiply by 4 to get byte count.

Protocol

IP Protocol ID. Including (but not limited to):

1	ICMP	17	UDP	57	SKIP
2	IGMP	47	GRE	88	EIGRP
6	TCP	50	ESP	89	OSPF
9	IGRP	51	AH	115	L2TP

Total Length

Total length of IP datagram, or IP fragment if fragmented. Measured in Bytes.

Fragment Offset

Fragment offset from start of IP datagram. Measured in 8 byte (2 words, 64 bits) increments. If IP datagram is fragmented, fragment size (Total Length) must be a multiple of 8 bytes.

Header Checksum

Checksum of entire IP header

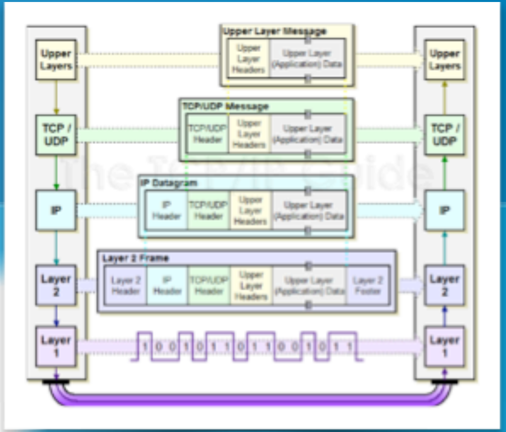
IP Flags

x D M

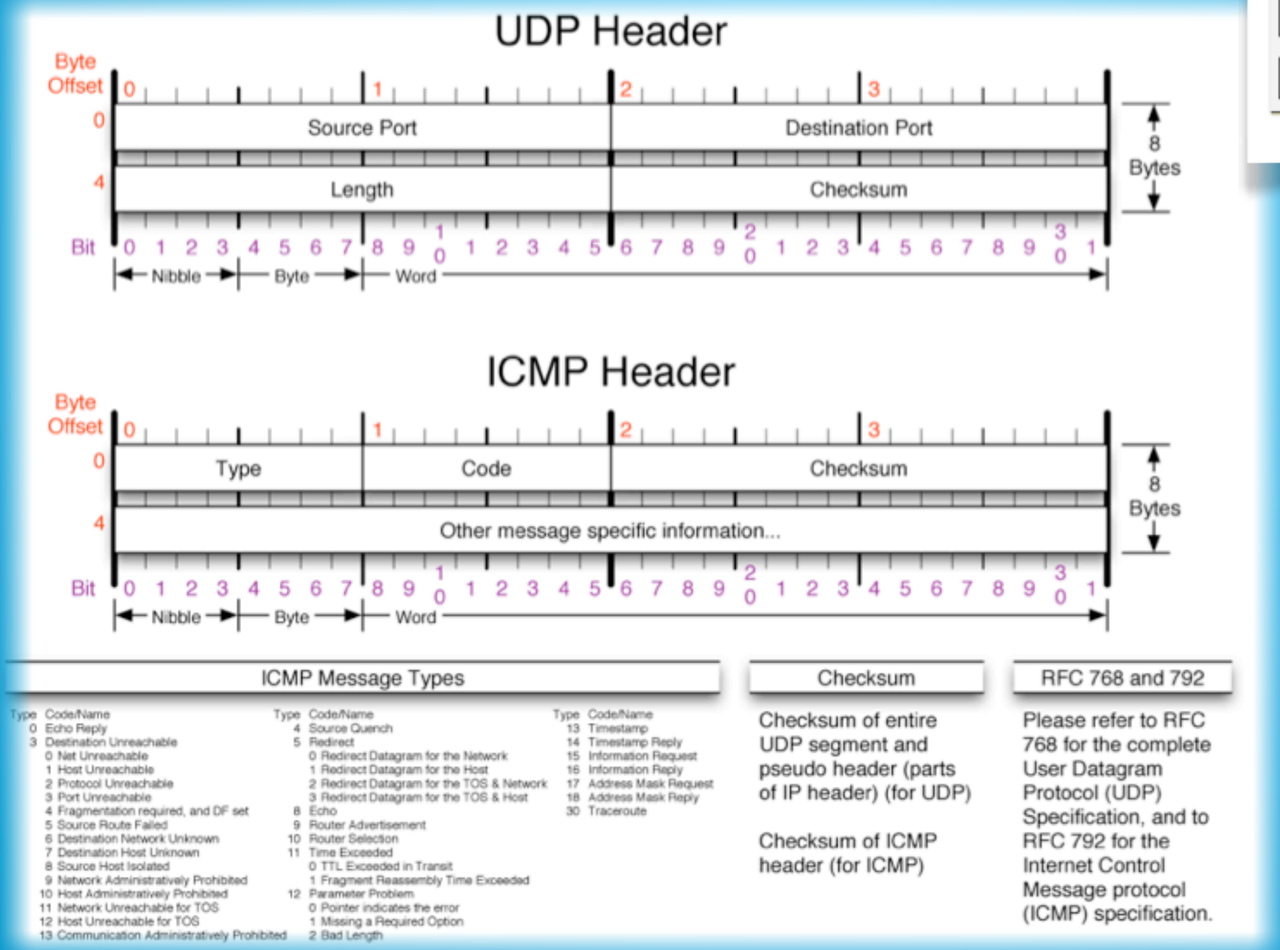
x 0x80 reserved (evil bit)
D 0x40 Do Not Fragment
M 0x20 More Fragments follow

RFC 791

Please refer to RFC 791 for the complete Internet Protocol (IP) Specification.



TCP



UDP Header

Byte Offset	Bits 0–15	Bits 16–31
0	Source Port	Destination Port
4	Length	Checksum

ICMP Header

Byte Offset	Bits 0–7	Bits 8–15	Bits 16–31
0	Type	Code	Checksum
4	Other message specific information...		

ICMP Message Types

Type	Code/Name
0	Echo Reply
3	Destination Unreachable
	0 Net Unreachable
	1 Host Unreachable
	2 Protocol Unreachable
	3 Port Unreachable
	4 Fragmentation required, and DF set
	5 Source Route Failed
	6 Destination Network Unknown
	7 Destination Host Unknown
	8 Source Host Isolated
	9 Network Administratively Prohibited
	10 Host Administratively Prohibited
	11 Network Unreachable for TOS
	12 Host Unreachable for TOS
	13 Communication Administratively Prohibited
4	Source Quench
5	Redirect
	0 Redirect Datagram for the Network
	1 Redirect Datagram for the Host
	2 Redirect Datagram for the TOS & Network
	3 Redirect Datagram for the TOS & Host
8	Echo
9	Router Advertisement
10	Router Selection
11	Time Exceeded
	0 TTL Exceeded in Transit
	1 Fragment Reassembly Time Exceeded
12	Parameter Problem
	0 Pointer indicates the error
	1 Missing a Required Option
	2 Bad Length
13	Timestamp
14	Timestamp Reply
15	Information Request
16	Information Reply
17	Address Mask Request
18	Address Mask Reply
30	Traceroute

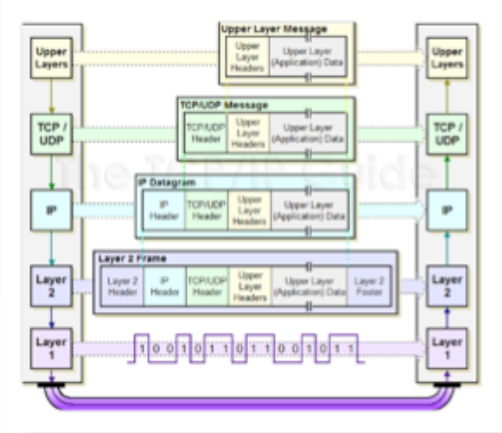
Checksum

Checksum of entire UDP segment and pseudo header (parts of IP header) (for UDP)

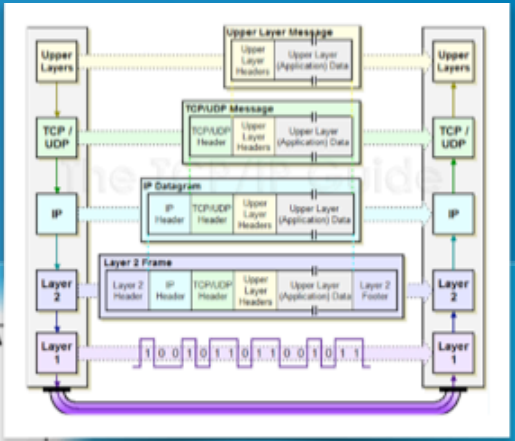
Checksum of ICMP header (for ICMP)

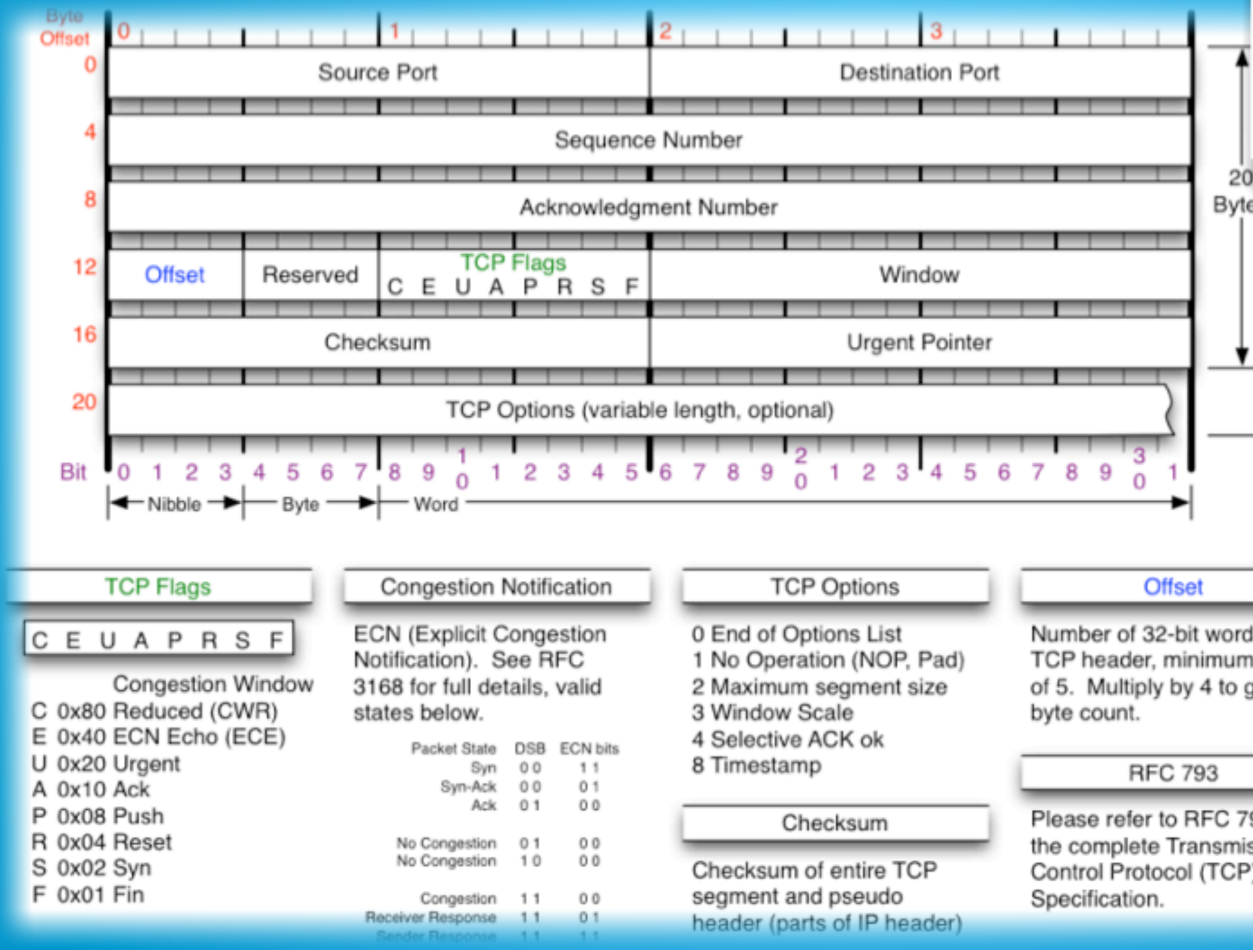
RFC 768 and 792

Please refer to RFC 768 for the complete User Datagram Protocol (UDP) Specification, and to RFC 792 for the Internet Control Message protocol (ICMP) specification.



TCP





Byte Offset	0	1	2	3
0	Source Port		Destination Port	
4	Sequence Number			
8	Acknowledgment Number			
12	Offset | Reserved	TCP Flags C E U A P R S F	Window	
16	Checksum		Urgent Pointer	
20	TCP Options (variable length, optional)			

Bit 0 1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 20 1 2 3 4 5 6 7 8 9 30 1

Nibble — Byte — Word

20 Bytes

Offset

TCP Flags

C E U A P R S F

Congestion Window
C 0x80 Reduced (CWR)
E 0x40 ECN Echo (ECE)
U 0x20 Urgent
A 0x10 Ack
P 0x08 Push
R 0x04 Reset
S 0x02 Syn
F 0x01 Fin

Congestion Notification

ECN (Explicit Congestion Notification). See RFC 3168 for full details, valid states below.

Packet State	DSB	ECN bits
Syn	0 0	1 1
Syn-Ack	0 0	0 1
Ack	0 1	0 0
No Congestion	0 1	0 0
No Congestion	1 0	0 0
Congestion	1 1	0 0
Receiver Response	1 1	0 1
Sender Response	1 1	1 1

TCP Options

0 End of Options List
1 No Operation (NOP, Pad)
2 Maximum segment size
3 Window Scale
4 Selective ACK ok
8 Timestamp

Checksum

Checksum of entire TCP segment and pseudo header (parts of IP header)

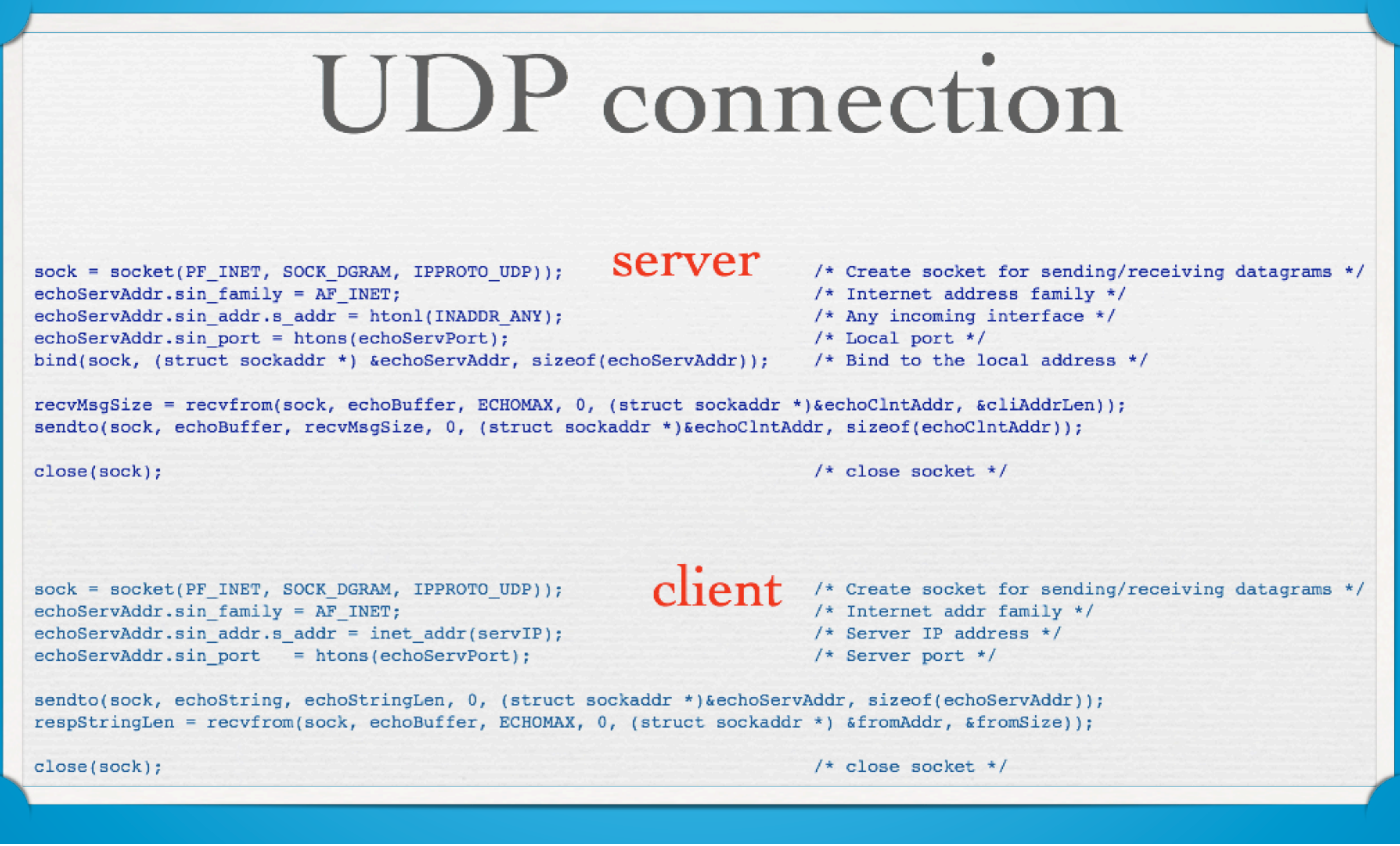
Offset

Number of 32-bit words in TCP header, minimum value of 5. Multiply by 4 to get byte count.

RFC 793

Please refer to RFC 793 for the complete Transmission Control Protocol (TCP) Specification.

TCP



UDP connection

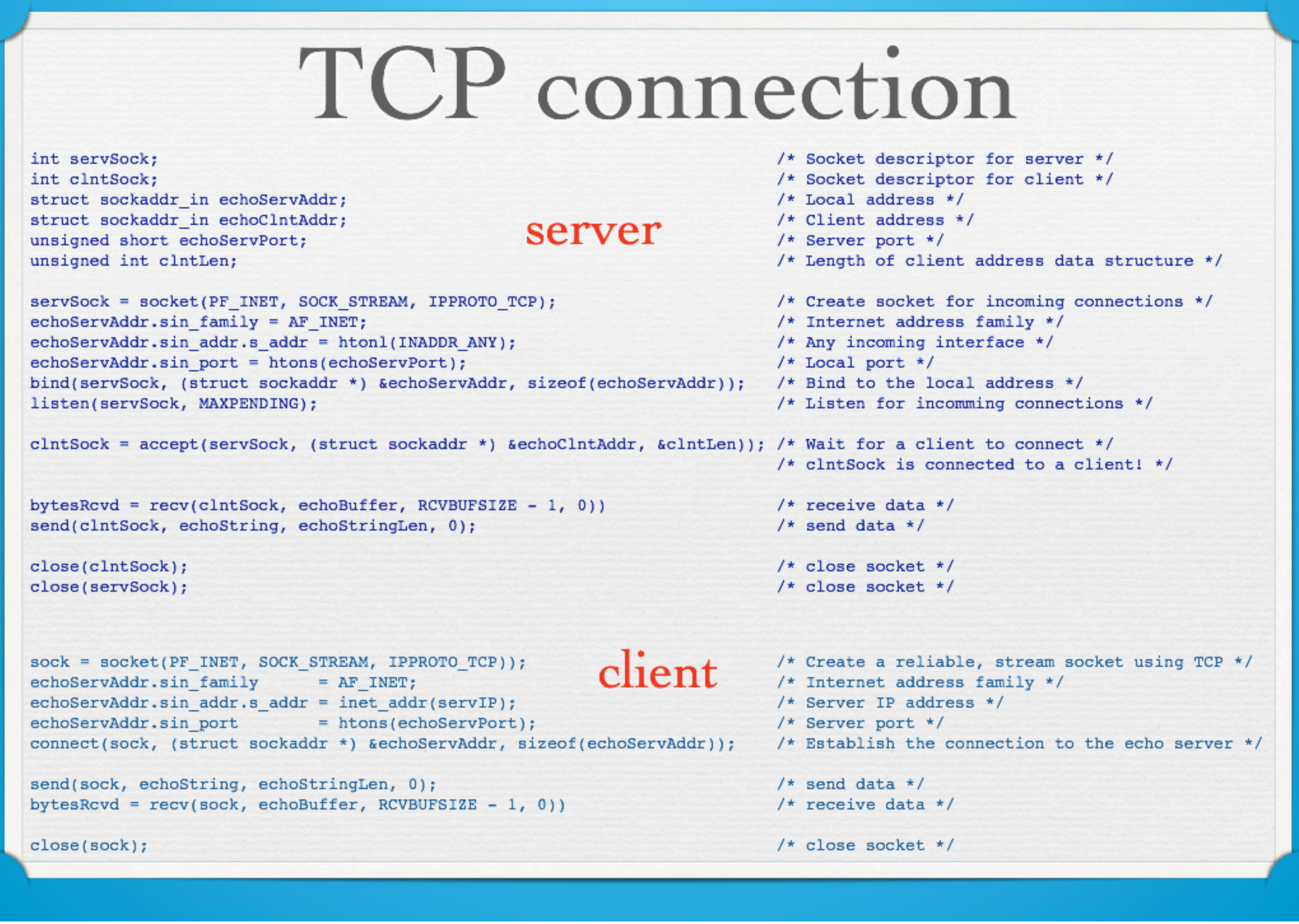
server

```
sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP));                /* Create socket for sending/receiving datagrams */
echoServAddr.sin_family = AF_INET;                               /* Internet address family */
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY);                /* Any incoming interface */
echoServAddr.sin_port = htons(echoServPort);                     /* Local port */
bind(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr));   /* Bind to the local address */

recvMsgSize = recvfrom(sock, echoBuffer, ECHOMAX, 0, (struct sockaddr *)&echoClntAddr, &cliAddrLen));
sendto(sock, echoBuffer, recvMsgSize, 0, (struct sockaddr *)&echoClntAddr, sizeof(echoClntAddr));

close(sock);                                                     /* close socket */
```

client

```
sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP));                /* Create socket for sending/receiving datagrams */
echoServAddr.sin_family = AF_INET;                               /* Internet addr family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP);                /* Server IP address */
echoServAddr.sin_port    = htons(echoServPort);                  /* Server port */

sendto(sock, echoString, echoStringLen, 0, (struct sockaddr *)&echoServAddr, sizeof(echoServAddr));
respStringLen = recvfrom(sock, echoBuffer, ECHOMAX, 0, (struct sockaddr *) &fromAddr, &fromSize));

close(sock);                                                     /* close socket */
```

TCP



TCP connection

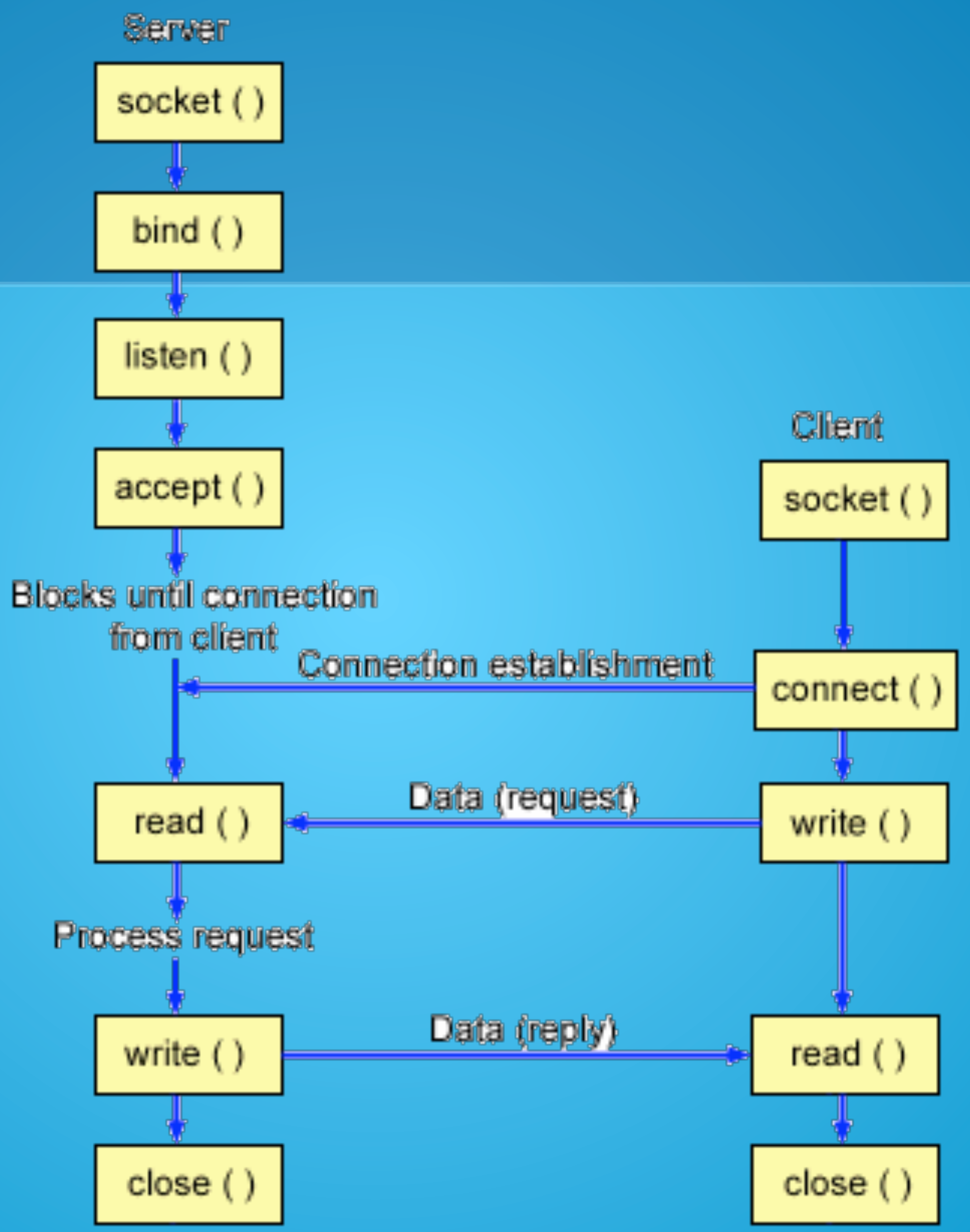
server

```
int servSock;                                                              /* Socket descriptor for server */
int clntSock;                                                              /* Socket descriptor for client */
struct sockaddr_in echoServAddr;                                           /* Local address */
struct sockaddr_in echoClntAddr;                                           /* Client address */
unsigned short echoServPort;                                               /* Server port */
unsigned int clntLen;                                                      /* Length of client address data structure */

servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);                      /* Create socket for incoming connections */
echoServAddr.sin_family = AF_INET;                                         /* Internet address family */
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY);                          /* Any incoming interface */
echoServAddr.sin_port = htons(echoServPort);                               /* Local port */
bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr));   /* Bind to the local address */
listen(servSock, MAXPENDING);                                              /* Listen for incomming connections */

clntSock = accept(servSock, (struct sockaddr *) &echoClntAddr, &clntLen)); /* Wait for a client to connect */
                                                                           /* clntSock is connected to a client! */

bytesRcvd = recv(clntSock, echoBuffer, RCVBUFSIZE - 1, 0))                 /* receive data */
send(clntSock, echoString, echoStringLen, 0);                              /* send data */

close(clntSock);                                                           /* close socket */
close(servSock);                                                           /* close socket */
```

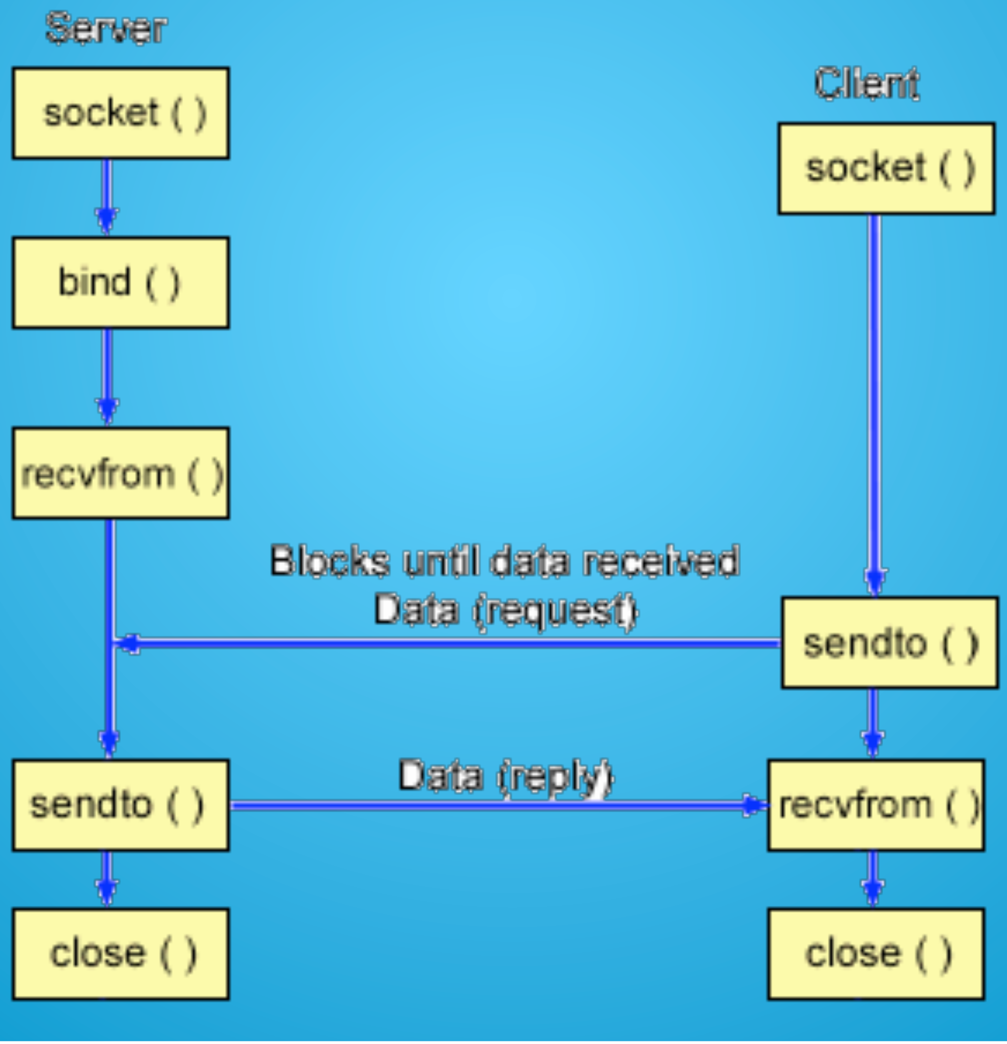
client

```
sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP));                         /* Create a reliable, stream socket using TCP */
echoServAddr.sin_family      = AF_INET;                                    /* Internet address family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP);                          /* Server IP address */
echoServAddr.sin_port        = htons(echoServPort);                        /* Server port */
connect(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr));    /* Establish the connection to the echo server */

send(sock, echoString, echoStringLen, 0);                                  /* send data */
bytesRcvd = recv(sock, echoBuffer, RCVBUFSIZE - 1, 0))                     /* receive data */

close(sock);                                                               /* close socket */
```

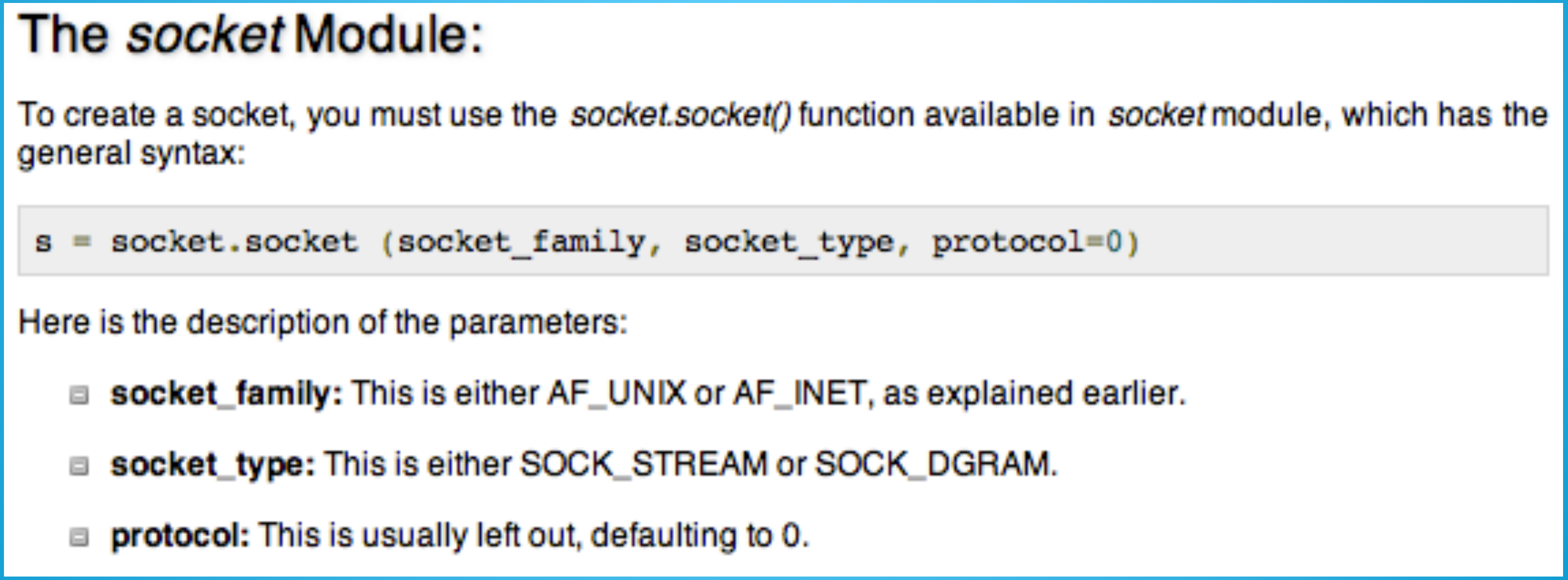
TCP


Server
socket ( )
bind ( )
listen ( )
accept ( )
Blocks until connection from client
Client
socket ( )
Connection establishment
connect ( )
read ( )
Data (request)
write ( )
Process request
write ( )
Data (reply)
read ( )
close ( )
close ( )

UDP


Server
socket ( )
bind ( )
recvfrom ( )
Blocks until data received
Data (request)
sendto ( )
close ( )
Client
socket ( )
sendto ( )
Data (reply)
recvfrom ( )
close ( )

Socket



The *socket* Module:

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

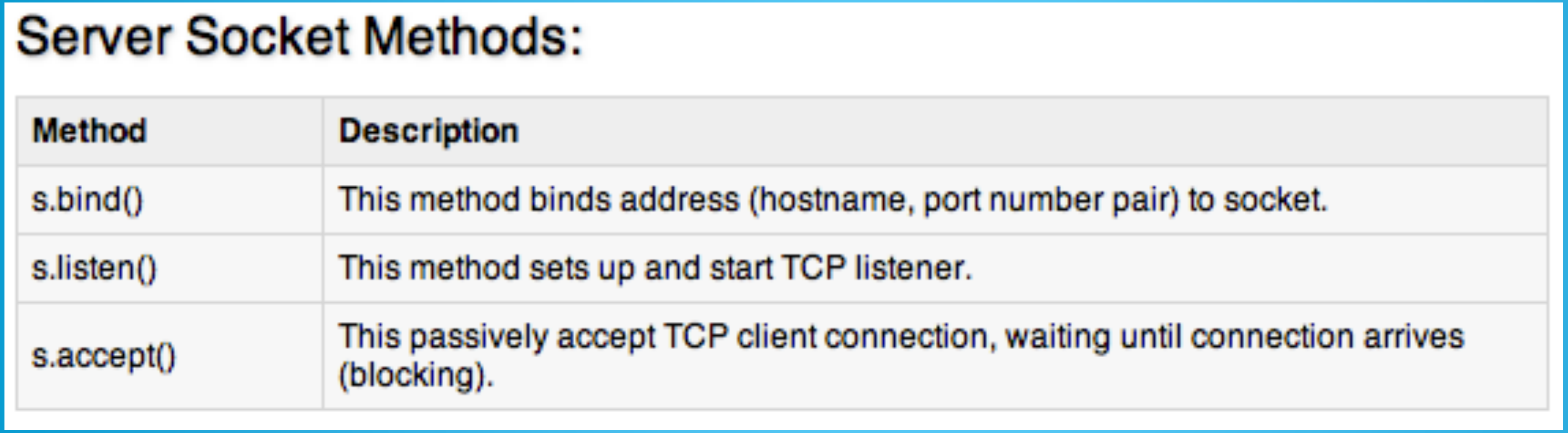
```
s = socket.socket (socket_family, socket_type, protocol=0)
```

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.

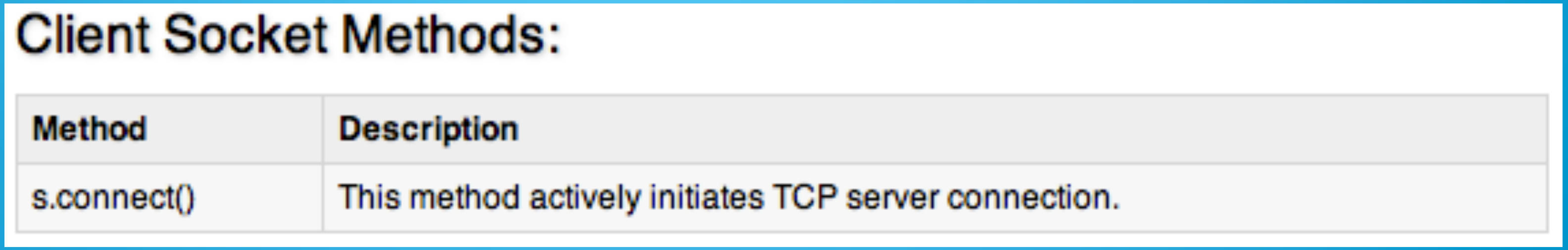
http://www.tutorialspoint.com/python/python_networking.htm

Socket



Server Socket Methods:

Method	Description
s.bind()	This method binds address (hostname, port number pair) to socket.
s.listen()	This method sets up and start TCP listener.
s.accept()	This passively accept TCP client connection, waiting until connection arrives (blocking).

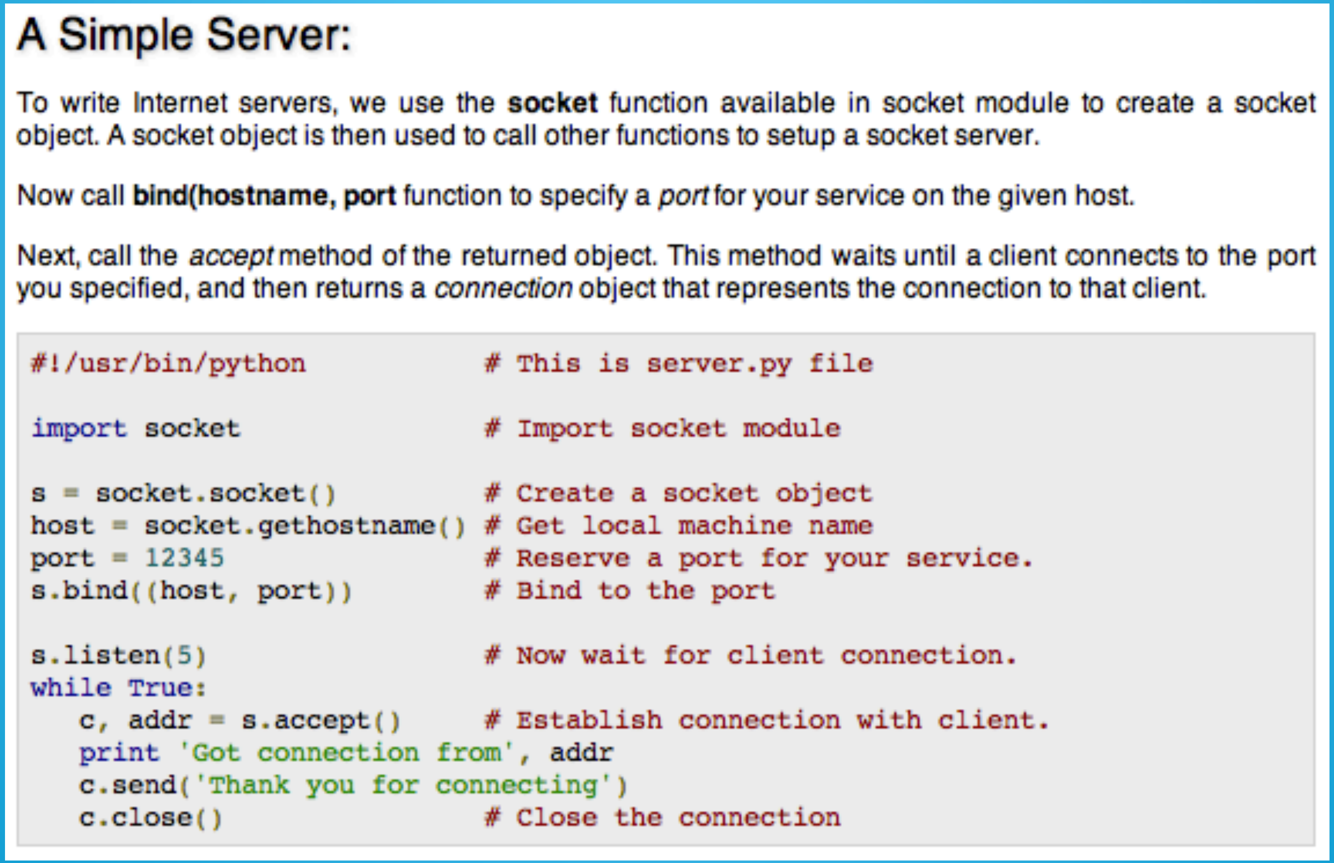


Client Socket Methods:

Method	Description
s.connect()	This method actively initiates TCP server connection.

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

Socket



A Simple Server:

To write Internet servers, we use the **socket** function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server.

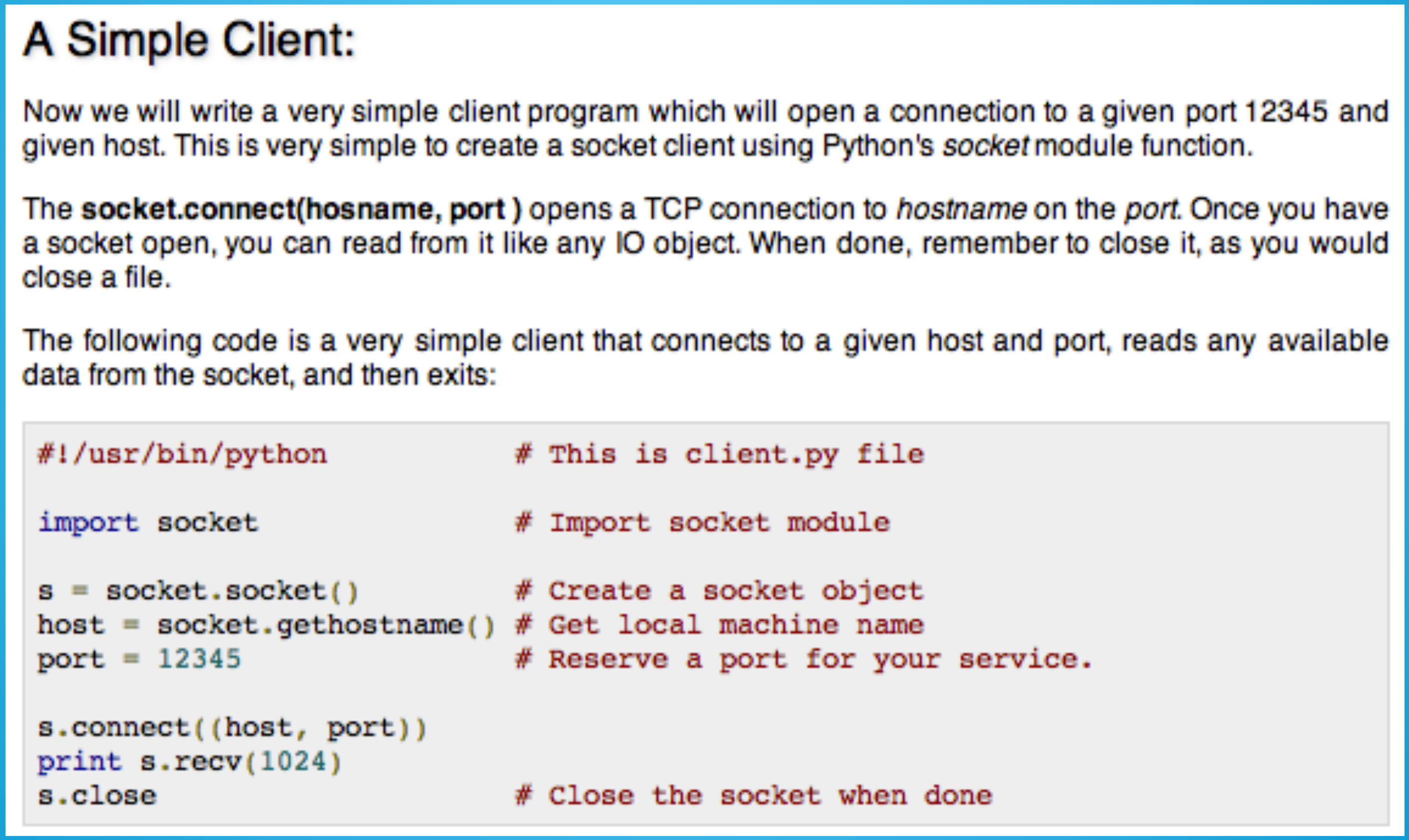
Now call **bind(hostname, port** function to specify a *port* for your service on the given host.

Next, call the *accept* method of the returned object. This method waits until a client connects to the port you specified, and then returns a *connection* object that represents the connection to that client.

```
#!/usr/bin/python           # This is server.py file

import socket               # Import socket module

s = socket.socket()         # Create a socket object
host = socket.gethostname() # Get local machine name
port = 12345                # Reserve a port for your service.
s.bind((host, port))        # Bind to the port

s.listen(5)                 # Now wait for client connection.
while True:
   c, addr = s.accept()     # Establish connection with client.
   print 'Got connection from', addr
   c.send('Thank you for connecting')
   c.close()                # Close the connection
```

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

Socket



A Simple Client:

Now we will write a very simple client program which will open a connection to a given port 12345 and given host. This is very simple to create a socket client using Python's *socket* module function.

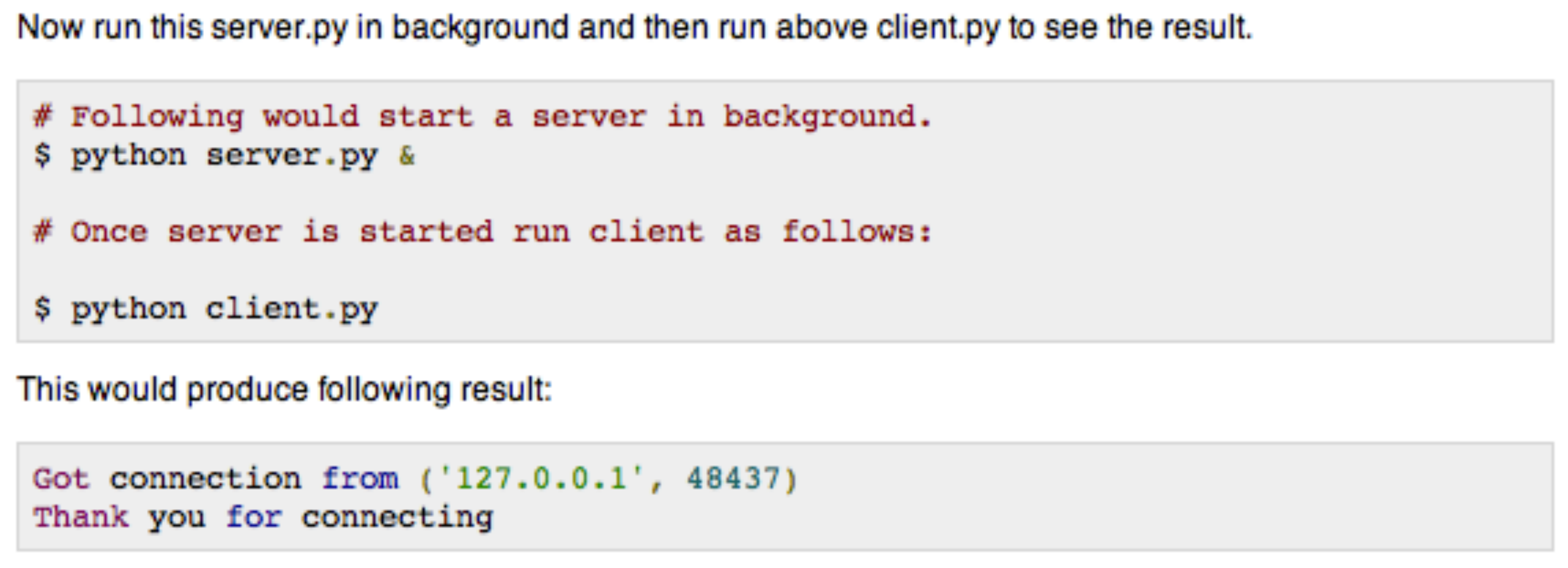
The **socket.connect(hosname, port)** opens a TCP connection to *hostname* on the *port*. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file.

The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits:

```
#!/usr/bin/python           # This is client.py file

import socket               # Import socket module

s = socket.socket()         # Create a socket object
host = socket.gethostname() # Get local machine name
port = 12345                # Reserve a port for your service.

s.connect((host, port))
print s.recv(1024)
s.close                     # Close the socket when done
```

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

Socket



Now run this server.py in background and then run above client.py to see the result.

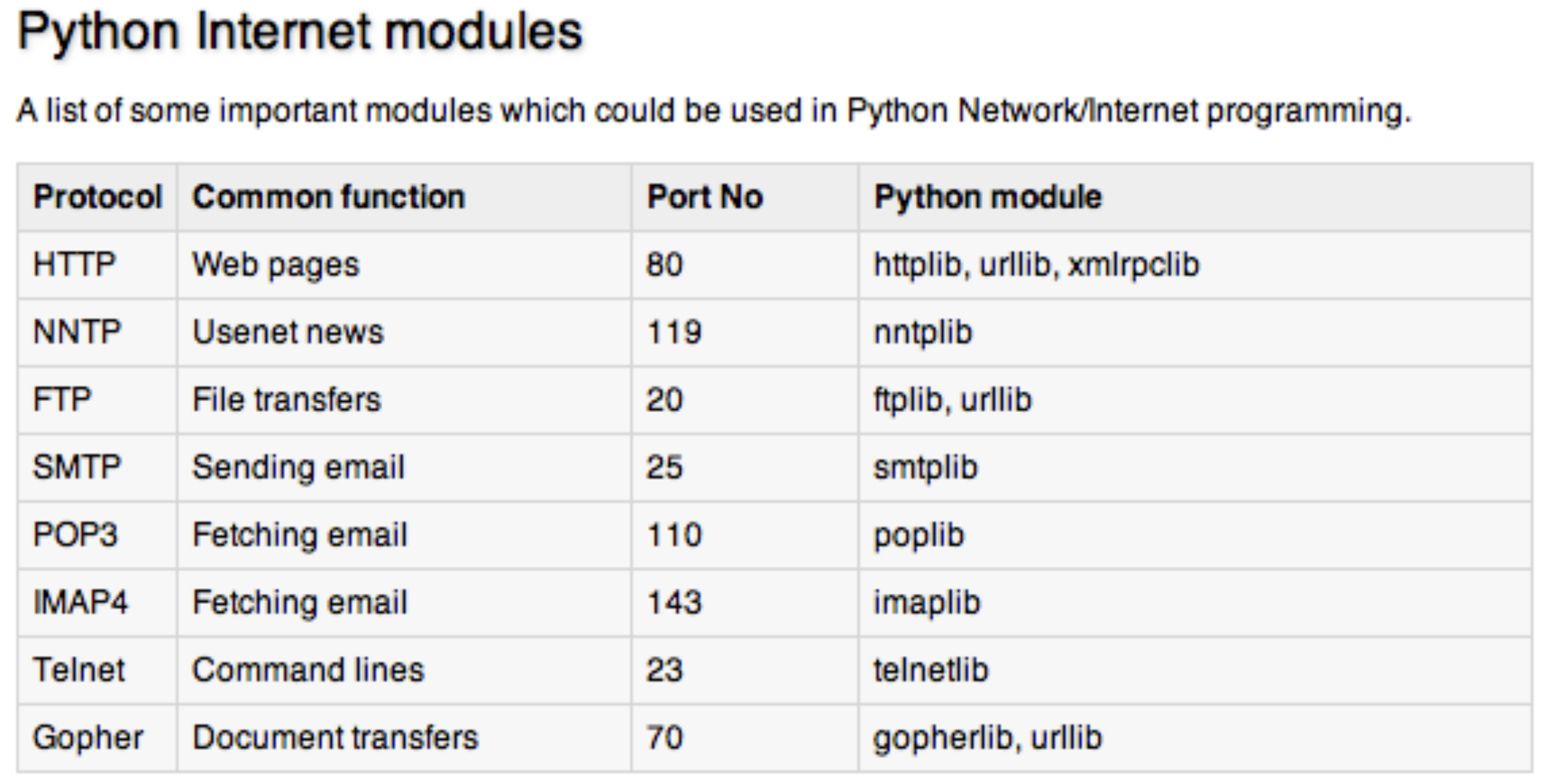
```
# Following would start a server in background.
$ python server.py &

# Once server is started run client as follows:

$ python client.py
```

This would produce following result:

```
Got connection from ('127.0.0.1', 48437)
Thank you for connecting
```

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

Socket



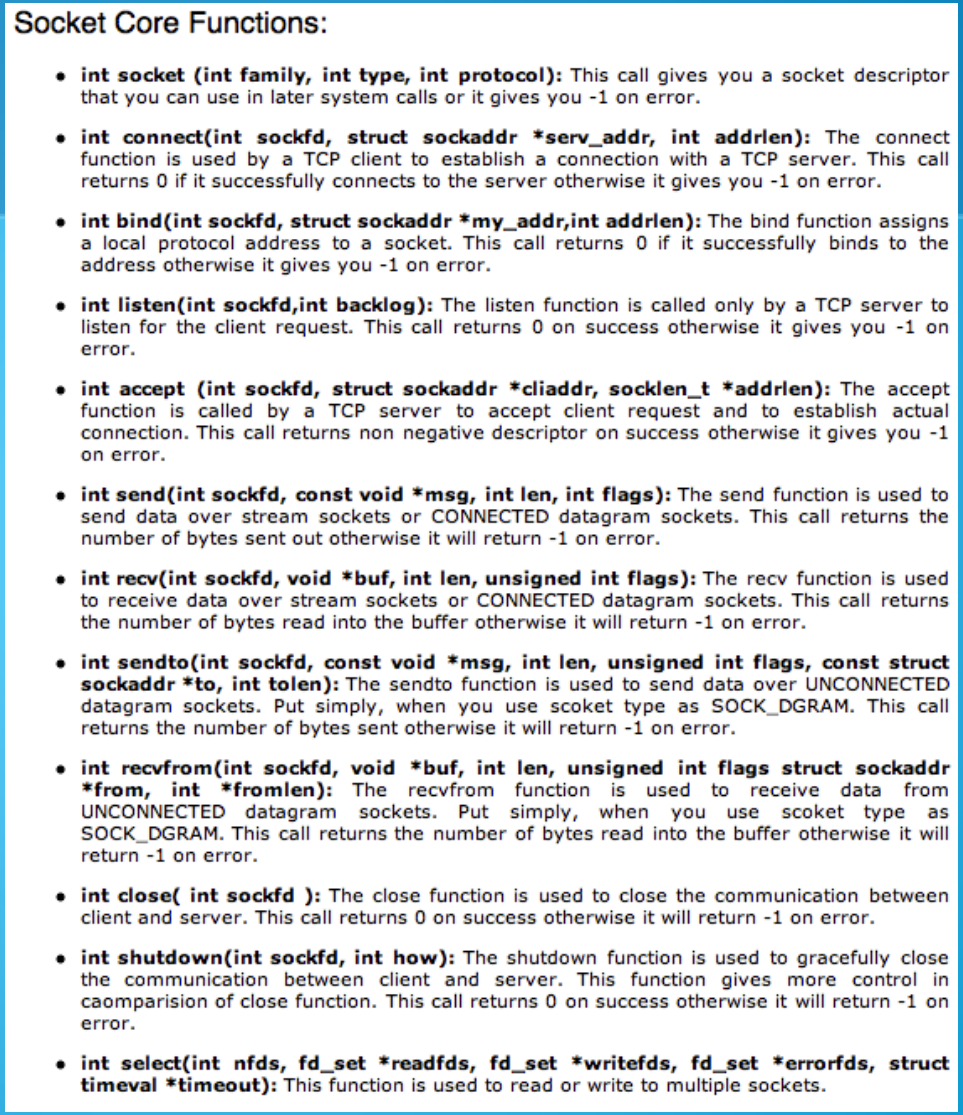
Python Internet modules

A list of some important modules which could be used in Python Network/Internet programming.

Protocol	Common function	Port No	Python module
HTTP	Web pages	80	httplib, urllib, xmlrpclib
NNTP	Usenet news	119	nntplib
FTP	File transfers	20	ftplib, urllib
SMTP	Sending email	25	smtplib
POP3	Fetching email	110	poplib
IMAP4	Fetching email	143	imaplib
Telnet	Command lines	23	telnetlib
Gopher	Document transfers	70	gopherlib, urllib

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

Socket

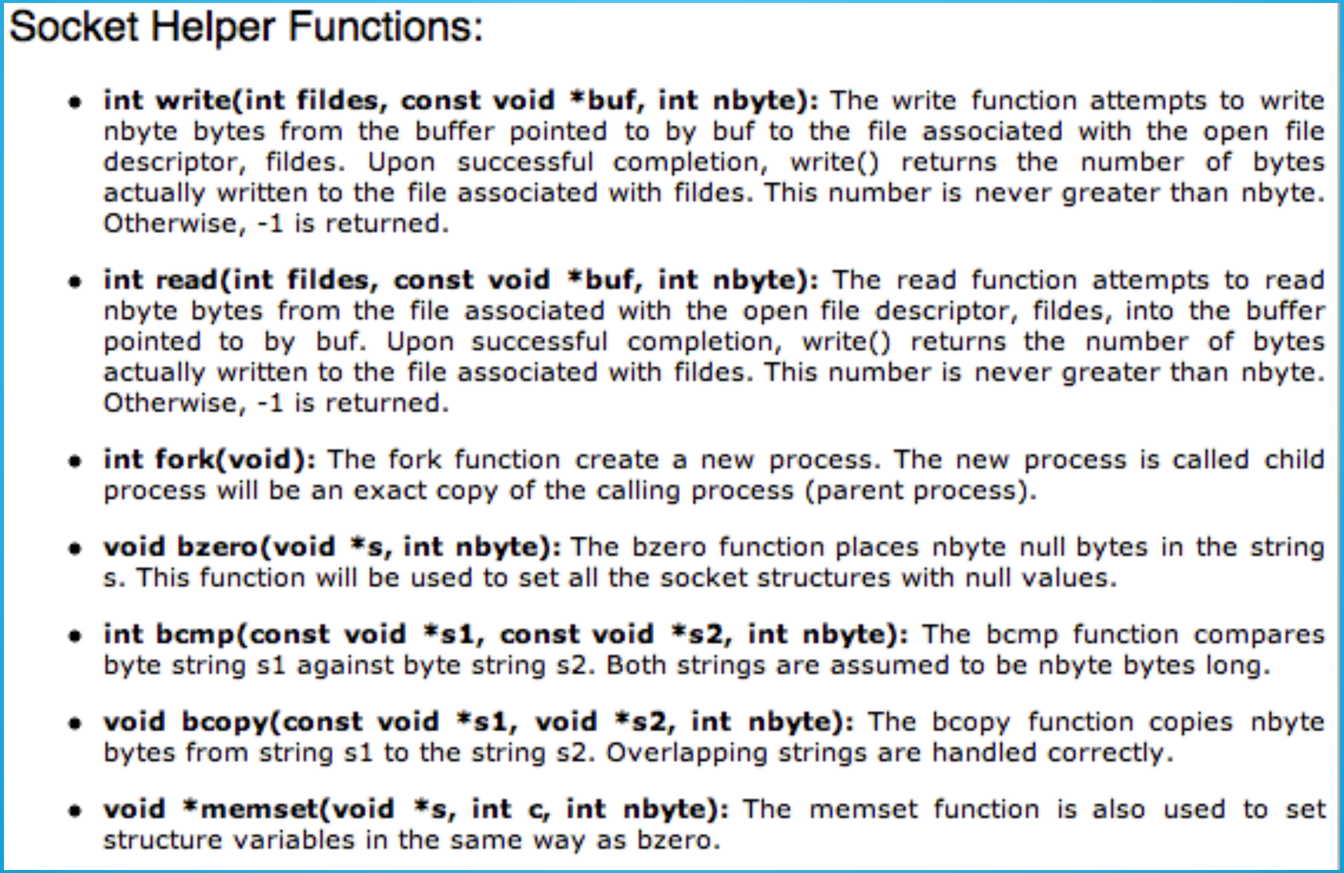


Socket Core Functions:

- **int socket (int family, int type, int protocol):** This call gives you a socket descriptor that you can use in later system calls or it gives you -1 on error.
- **int connect(int sockfd, struct sockaddr *serv_addr, int addrlen):** The connect function is used by a TCP client to establish a connection with a TCP server. This call returns 0 if it successfully connects to the server otherwise it gives you -1 on error.
- **int bind(int sockfd, struct sockaddr *my_addr,int addrlen):** The bind function assigns a local protocol address to a socket. This call returns 0 if it successfully binds to the address otherwise it gives you -1 on error.
- **int listen(int sockfd,int backlog):** The listen function is called only by a TCP server to listen for the client request. This call returns 0 on success otherwise it gives you -1 on error.
- **int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen):** The accept function is called by a TCP server to accept client request and to establish actual connection. This call returns non negative descriptor on success otherwise it gives you -1 on error.
- **int send(int sockfd, const void *msg, int len, int flags):** The send function is used to send data over stream sockets or CONNECTED datagram sockets. This call returns the number of bytes sent out otherwise it will return -1 on error.
- **int recv(int sockfd, void *buf, int len, unsigned int flags):** The recv function is used to receive data over stream sockets or CONNECTED datagram sockets. This call returns the number of bytes read into the buffer otherwise it will return -1 on error.
- **int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen):** The sendto function is used to send data over UNCONNECTED datagram sockets. Put simply, when you use scoket type as SOCK_DGRAM. This call returns the number of bytes sent otherwise it will return -1 on error.
- **int recvfrom(int sockfd, void *buf, int len, unsigned int flags struct sockaddr *from, int *fromlen):** The recvfrom function is used to receive data from UNCONNECTED datagram sockets. Put simply, when you use scoket type as SOCK_DGRAM. This call returns the number of bytes read into the buffer otherwise it will return -1 on error.
- **int close(int sockfd):** The close function is used to close the communication between client and server. This call returns 0 on success otherwise it will return -1 on error.
- **int shutdown(int sockfd, int how):** The shutdown function is used to gracefully close the communication between client and server. This function gives more control in caomparision of close function. This call returns 0 on success otherwise it will return -1 on error.
- **int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout):** This function is used to read or write to multiple sockets.

http://www.tutorialspoint.com/unix_sockets/socket_quick_guide.htm

Socket

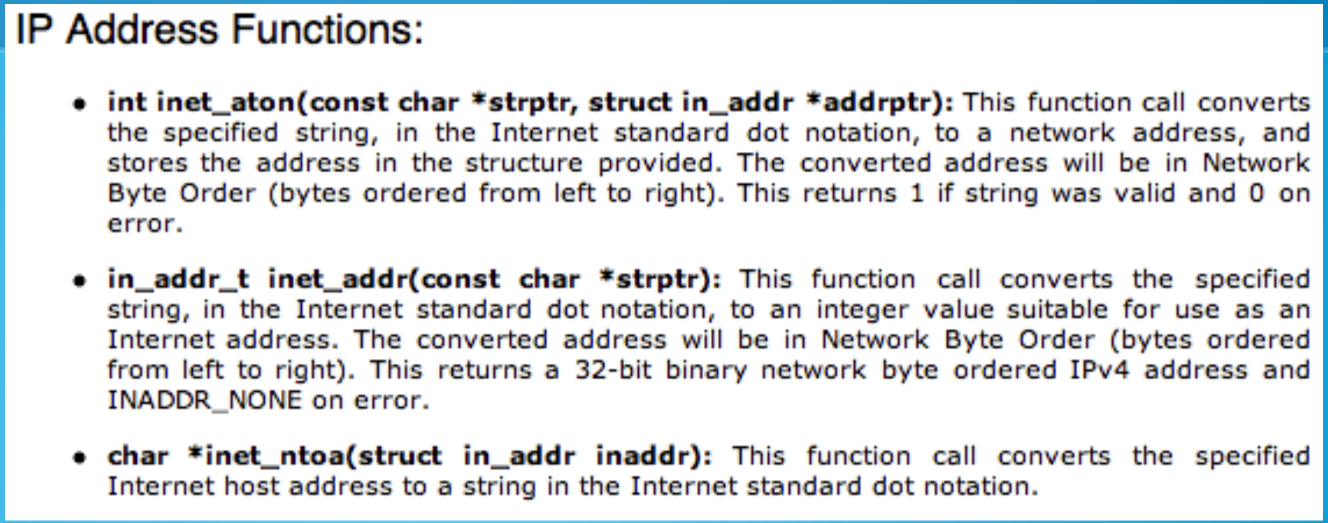


Socket Helper Functions:

- **int write(int fildes, const void *buf, int nbyte):** The write function attempts to write nbyte bytes from the buffer pointed to by buf to the file associated with the open file descriptor, fildes. Upon successful completion, write() returns the number of bytes actually written to the file associated with fildes. This number is never greater than nbyte. Otherwise, -1 is returned.
- **int read(int fildes, const void *buf, int nbyte):** The read function attempts to read nbyte bytes from the file associated with the open file descriptor, fildes, into the buffer pointed to by buf. Upon successful completion, write() returns the number of bytes actually written to the file associated with fildes. This number is never greater than nbyte. Otherwise, -1 is returned.
- **int fork(void):** The fork function create a new process. The new process is called child process will be an exact copy of the calling process (parent process).
- **void bzero(void *s, int nbyte):** The bzero function places nbyte null bytes in the string s. This function will be used to set all the socket structures with null values.
- **int bcmp(const void *s1, const void *s2, int nbyte):** The bcmp function compares byte string s1 against byte string s2. Both strings are assumed to be nbyte bytes long.
- **void bcopy(const void *s1, void *s2, int nbyte):** The bcopy function copies nbyte bytes from string s1 to the string s2. Overlapping strings are handled correctly.
- **void *memset(void *s, int c, int nbyte):** The memset function is also used to set structure variables in the same way as bzero.

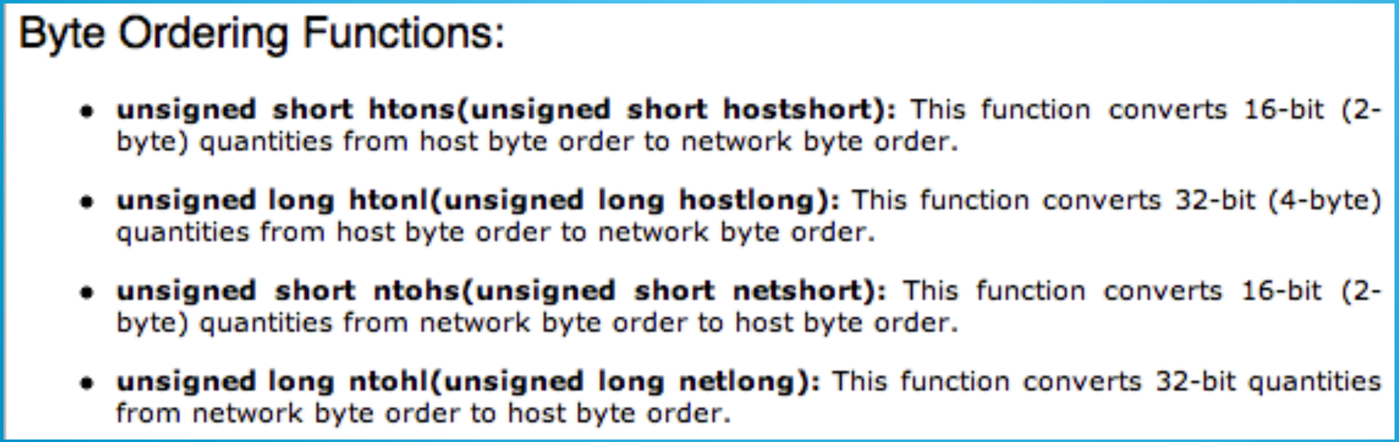
http://www.tutorialspoint.com/unix_sockets/socket_quick_guide.htm

Socket



IP Address Functions:

- **int inet_aton(const char *strptr, struct in_addr *addrptr):** This function call converts the specified string, in the Internet standard dot notation, to a network address, and stores the address in the structure provided. The converted address will be in Network Byte Order (bytes ordered from left to right). This returns 1 if string was valid and 0 on error.
- **in_addr_t inet_addr(const char *strptr):** This function call converts the specified string, in the Internet standard dot notation, to an integer value suitable for use as an Internet address. The converted address will be in Network Byte Order (bytes ordered from left to right). This returns a 32-bit binary network byte ordered IPv4 address and INADDR_NONE on error.
- **char *inet_ntoa(struct in_addr inaddr):** This function call converts the specified Internet host address to a string in the Internet standard dot notation.

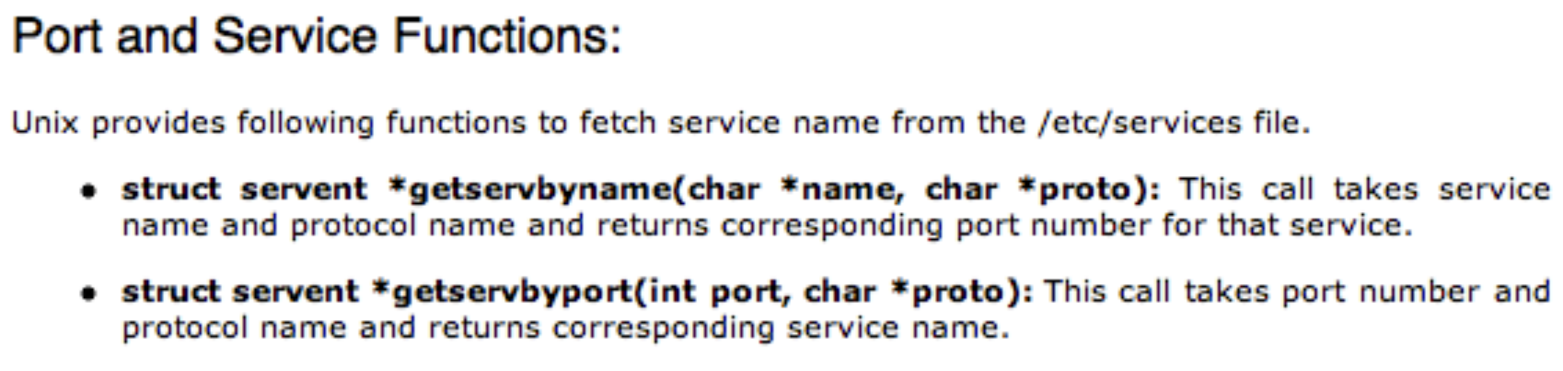


Byte Ordering Functions:

- **unsigned short htons(unsigned short hostshort):** This function converts 16-bit (2-byte) quantities from host byte order to network byte order.
- **unsigned long htonl(unsigned long hostlong):** This function converts 32-bit (4-byte) quantities from host byte order to network byte order.
- **unsigned short ntohs(unsigned short netshort):** This function converts 16-bit (2-byte) quantities from network byte order to host byte order.
- **unsigned long ntohl(unsigned long netlong):** This function converts 32-bit quantities from network byte order to host byte order.

http://www.tutorialspoint.com/unix_sockets/socket_quick_guide.htm

Socket



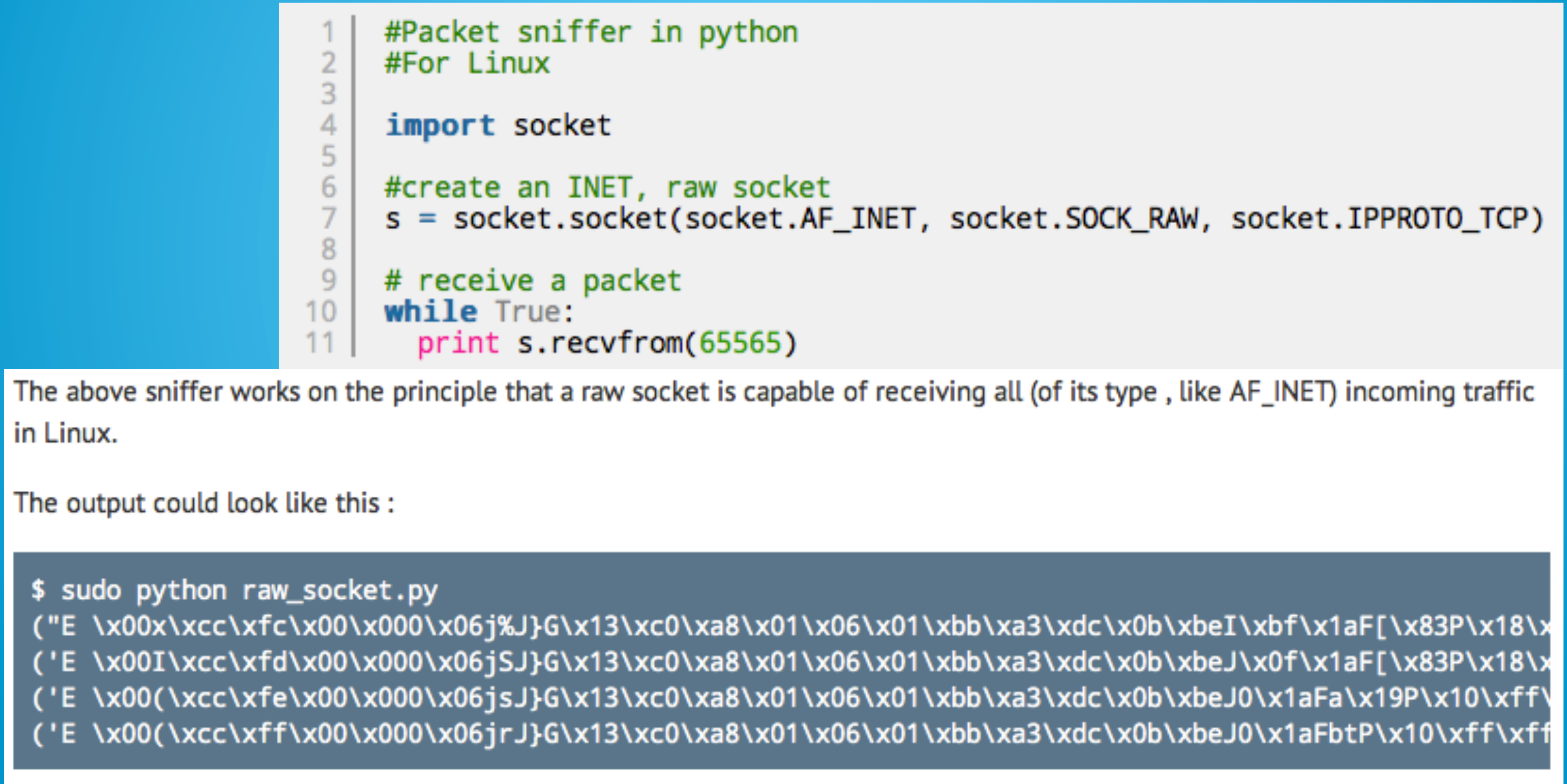
Port and Service Functions:

Unix provides following functions to fetch service name from the /etc/services file.

- **struct servent *getservbyname(char *name, char *proto):** This call takes service name and protocol name and returns corresponding port number for that service.
- **struct servent *getservbyport(int port, char *proto):** This call takes port number and protocol name and returns corresponding service name.

http://www.tutorialspoint.com/unix_sockets/socket_quick_guide.htm

Socket



```python
#Packet sniffer in python
#For Linux

import socket

#create an INET, raw socket
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_TCP)

# receive a packet
while True:
  print s.recvfrom(65565)
```

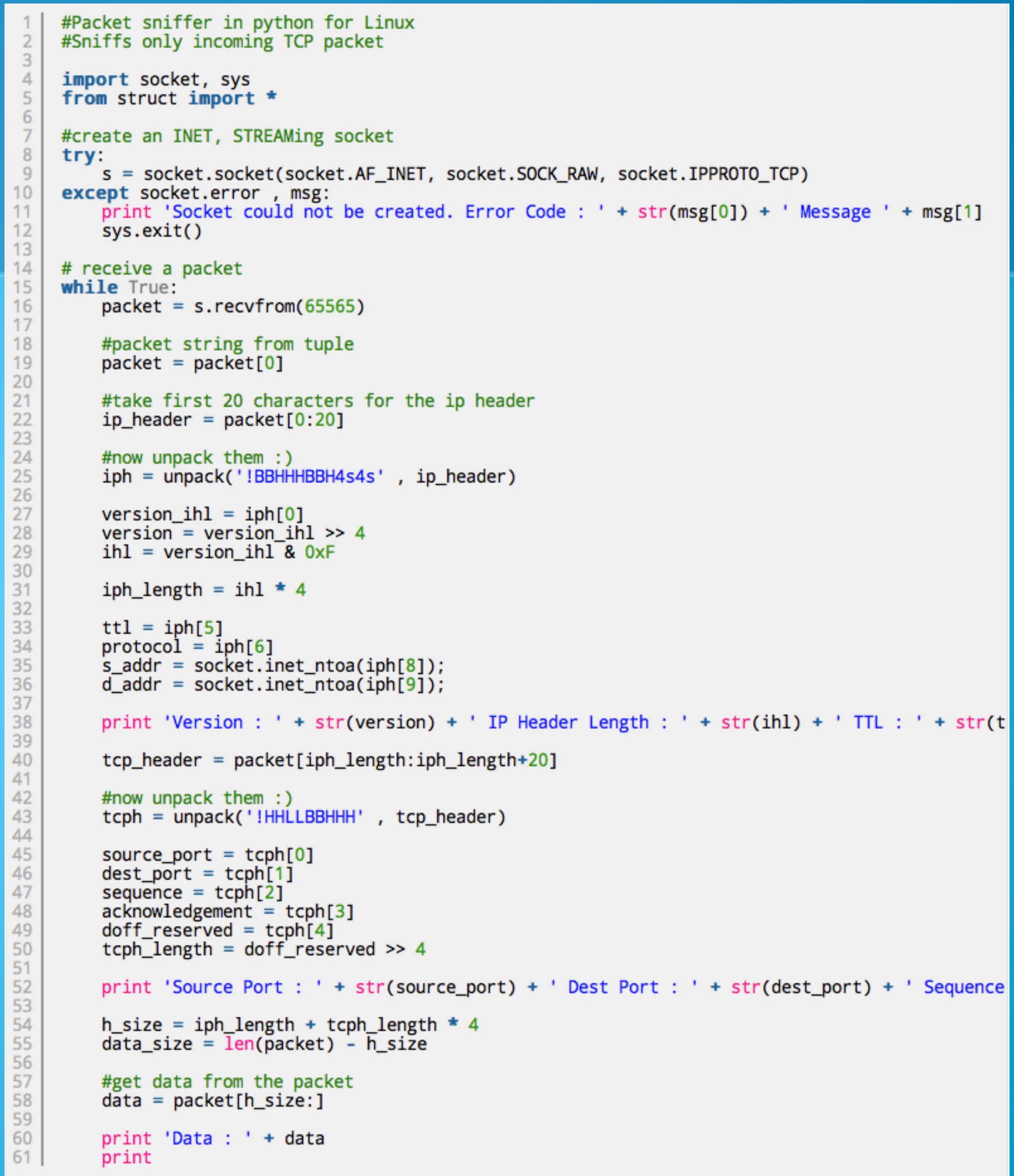
The above sniffer works on the principle that a raw socket is capable of receiving all (of its type , like AF_INET) incoming traffic in Linux.

The output could look like this :

```
$ sudo python raw_socket.py
("E \x00x\xcc\xfc\x00\x000\x06j%J}G\x13\xc0\xa8\x01\x06\x01\xbb\xa3\xdc\x0b\xbeI\xbf\x1aF[\x83P\x18\x
('E \x00I\xcc\xfd\x00\x000\x06jSJ}G\x13\xc0\xa8\x01\x06\x01\xbb\xa3\xdc\x0b\xbeJ\x0f\x1aF[\x83P\x18\x
('E \x00(\xcc\xfe\x00\x000\x06jsJ}G\x13\xc0\xa8\x01\x06\x01\xbb\xa3\xdc\x0b\xbeJ0\x1aFa\x19P\x10\xff\
('E \x00(\xcc\xff\x00\x000\x06jrJ}G\x13\xc0\xa8\x01\x06\x01\xbb\xa3\xdc\x0b\xbeJ0\x1aFbtP\x10\xff\xff
```

http://www.binarytides.com/python-packet-sniffer-code-linux/

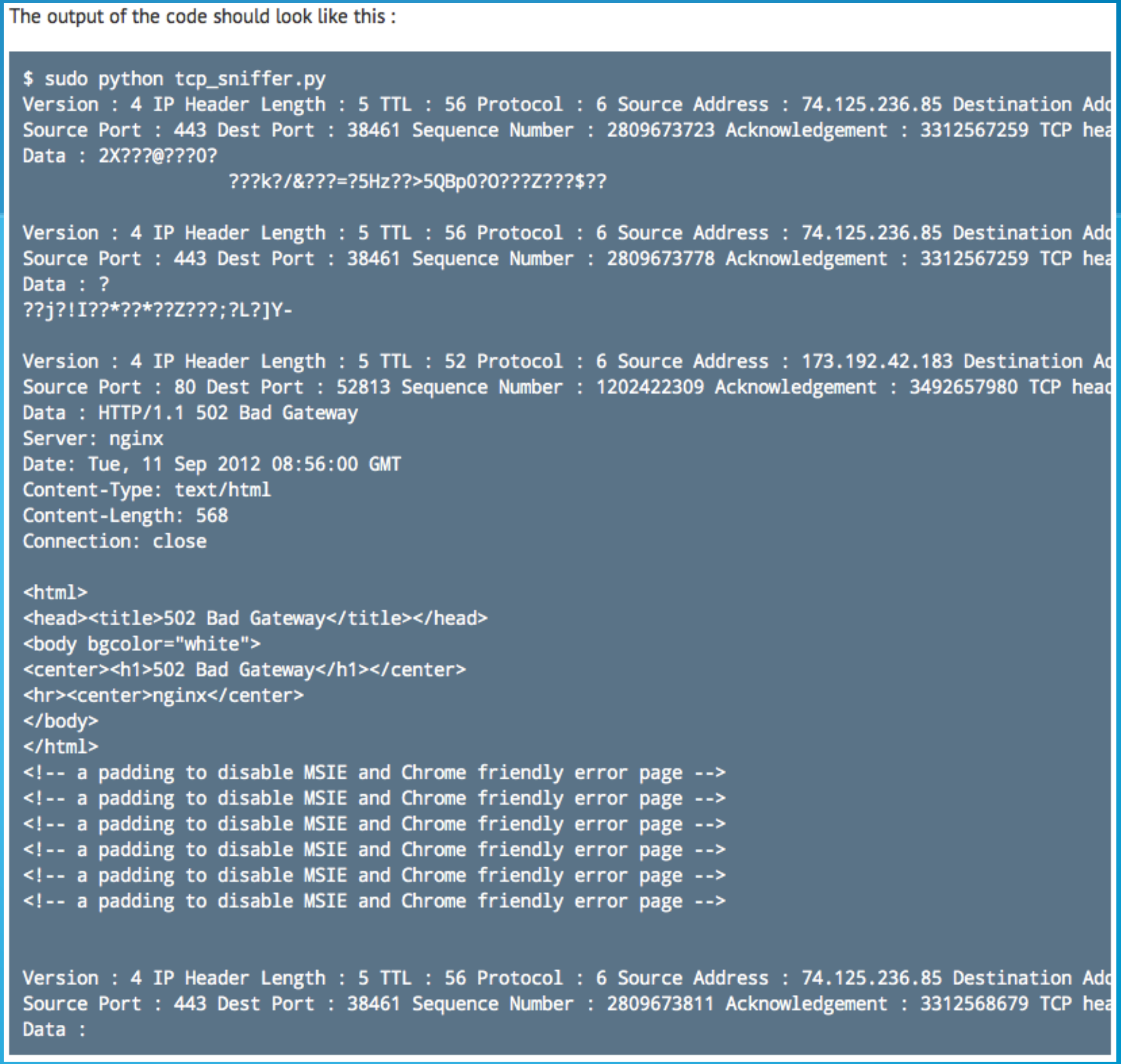
Socket



```python
#Packet sniffer in python for Linux
#Sniffs only incoming TCP packet

import socket, sys
from struct import *

#create an INET, STREAMing socket
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_TCP)
except socket.error , msg:
    print 'Socket could not be created. Error Code : ' + str(msg[0]) + ' Message ' + msg[1]
    sys.exit()

# receive a packet
while True:
    packet = s.recvfrom(65565)

    #packet string from tuple
    packet = packet[0]

    #take first 20 characters for the ip header
    ip_header = packet[0:20]

    #now unpack them :)
    iph = unpack('!BBHHHBBH4s4s' , ip_header)

    version_ihl = iph[0]
    version = version_ihl >> 4
    ihl = version_ihl & 0xF

    iph_length = ihl * 4

    ttl = iph[5]
    protocol = iph[6]
    s_addr = socket.inet_ntoa(iph[8]);
    d_addr = socket.inet_ntoa(iph[9]);

    print 'Version : ' + str(version) + ' IP Header Length : ' + str(ihl) + ' TTL : ' + str(t

    tcp_header = packet[iph_length:iph_length+20]

    #now unpack them :)
    tcph = unpack('!HHLLBBHHH' , tcp_header)

    source_port = tcph[0]
    dest_port = tcph[1]
    sequence = tcph[2]
    acknowledgement = tcph[3]
    doff_reserved = tcph[4]
    tcph_length = doff_reserved >> 4

    print 'Source Port : ' + str(source_port) + ' Dest Port : ' + str(dest_port) + ' Sequence

    h_size = iph_length + tcph_length * 4
    data_size = len(packet) - h_size

    #get data from the packet
    data = packet[h_size:]

    print 'Data : ' + data
    print
```

http://www.binarytides.com/python-packet-sniffer-code-linux/

Socket

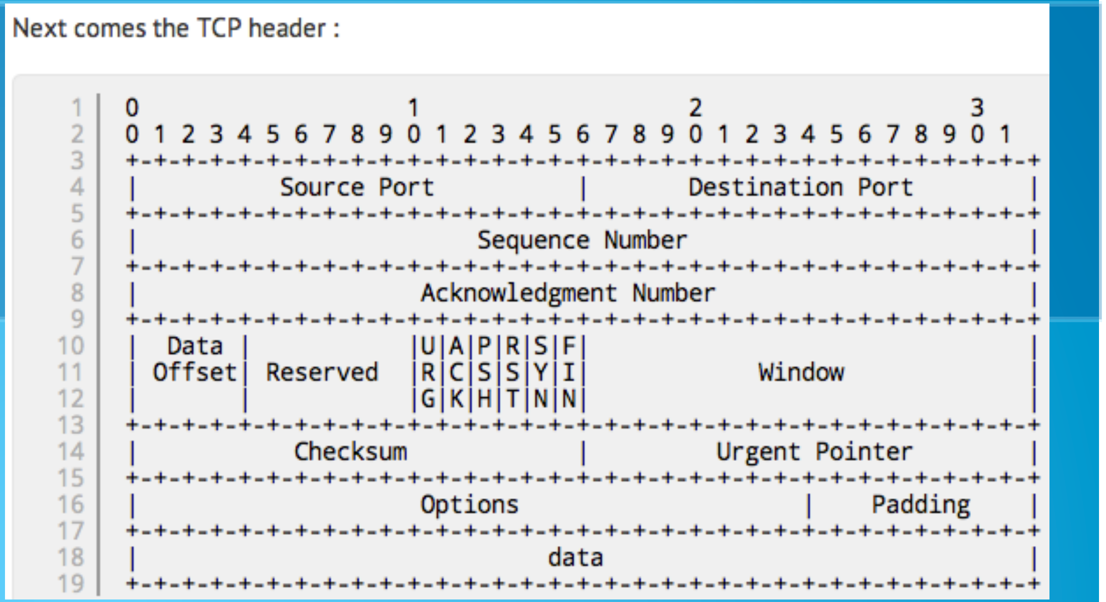


The output of the code should look like this :

```
$ sudo python tcp_sniffer.py
Version : 4 IP Header Length : 5 TTL : 56 Protocol : 6 Source Address : 74.125.236.85 Destination Add
Source Port : 443 Dest Port : 38461 Sequence Number : 2809673723 Acknowledgement : 3312567259 TCP hea
Data : 2X???@???0?
                   ???k?/&???=?5Hz??>5QBp0?0???Z???$??

Version : 4 IP Header Length : 5 TTL : 56 Protocol : 6 Source Address : 74.125.236.85 Destination Add
Source Port : 443 Dest Port : 38461 Sequence Number : 2809673778 Acknowledgement : 3312567259 TCP hea
Data : ?
??j?!I??*??*??Z???;?L?]Y-

Version : 4 IP Header Length : 5 TTL : 52 Protocol : 6 Source Address : 173.192.42.183 Destination Ad
Source Port : 80 Dest Port : 52813 Sequence Number : 1202422309 Acknowledgement : 3492657980 TCP head
Data : HTTP/1.1 502 Bad Gateway
Server: nginx
Date: Tue, 11 Sep 2012 08:56:00 GMT
Content-Type: text/html
Content-Length: 568
Connection: close

<html>
<head><title>502 Bad Gateway</title></head>
<body bgcolor="white">
<center><h1>502 Bad Gateway</h1></center>
<hr><center>nginx</center>
</body>
</html>
<!-- a padding to disable MSIE and Chrome friendly error page -->
<!-- a padding to disable MSIE and Chrome friendly error page -->
<!-- a padding to disable MSIE and Chrome friendly error page -->
<!-- a padding to disable MSIE and Chrome friendly error page -->
<!-- a padding to disable MSIE and Chrome friendly error page -->
<!-- a padding to disable MSIE and Chrome friendly error page -->


Version : 4 IP Header Length : 5 TTL : 56 Protocol : 6 Source Address : 74.125.236.85 Destination Add
Source Port : 443 Dest Port : 38461 Sequence Number : 2809673811 Acknowledgement : 3312568679 TCP hea
Data :
```

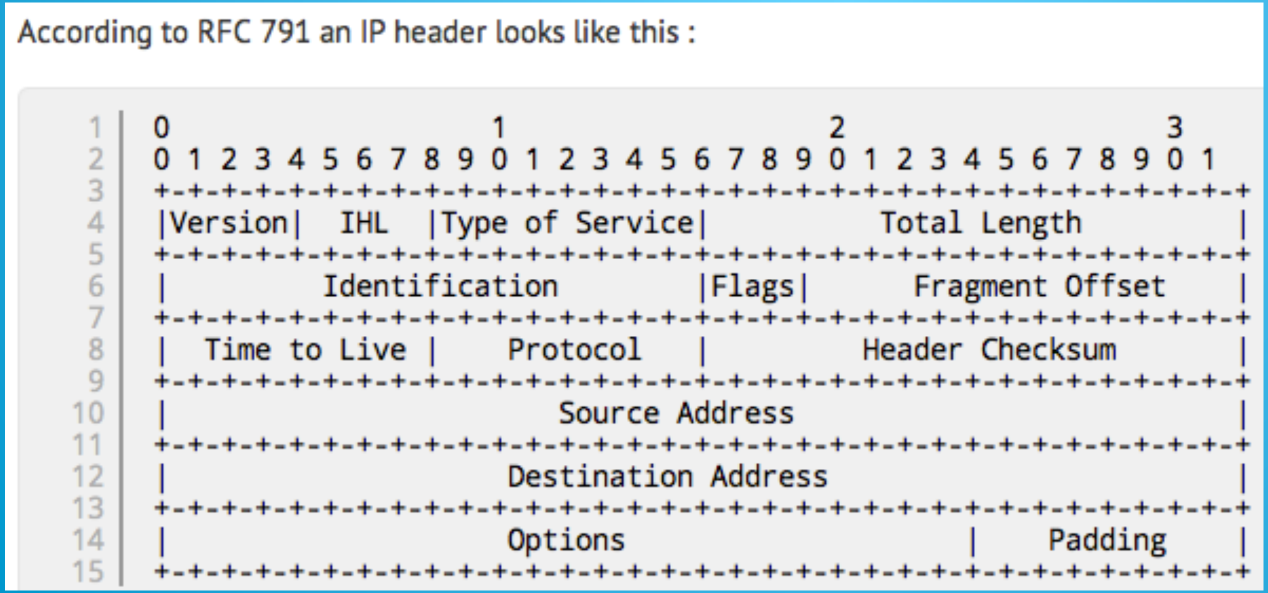
http://www.binarytides.com/python-packet-sniffer-code-linux/

Socket



Next comes the TCP header :

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

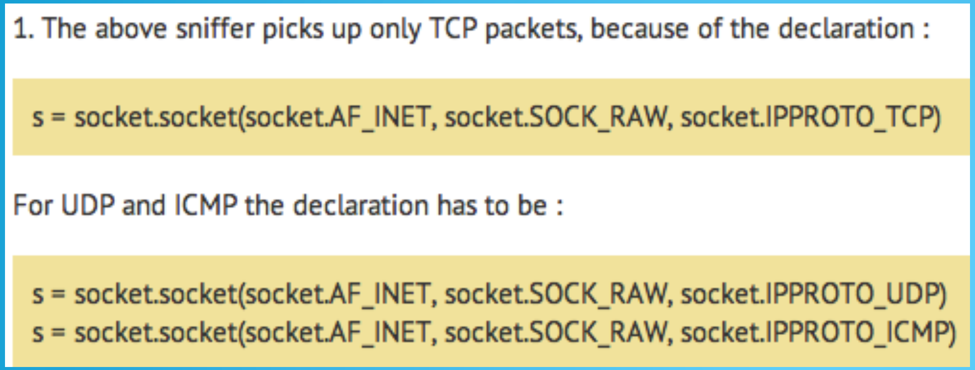


According to RFC 791 an IP header looks like this :

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

http://www.binarytides.com/python-packet-sniffer-code-linux/

Socket

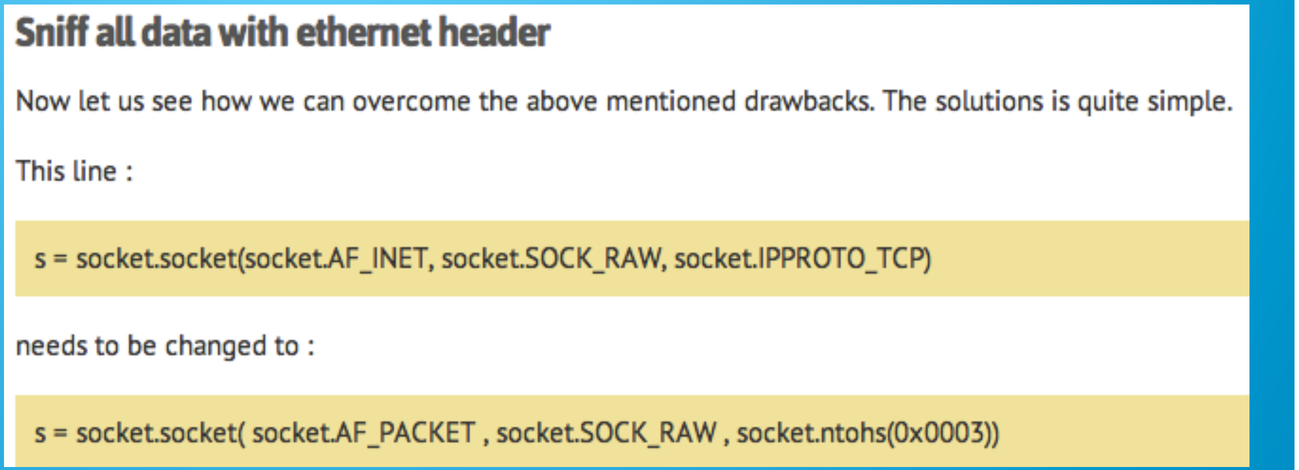


1. The above sniffer picks up only TCP packets, because of the declaration :

```
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_TCP)
```

For UDP and ICMP the declaration has to be :

```
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_UDP)
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)
```



Sniff all data with ethernet header

Now let us see how we can overcome the above mentioned drawbacks. The solutions is quite simple.

This line :

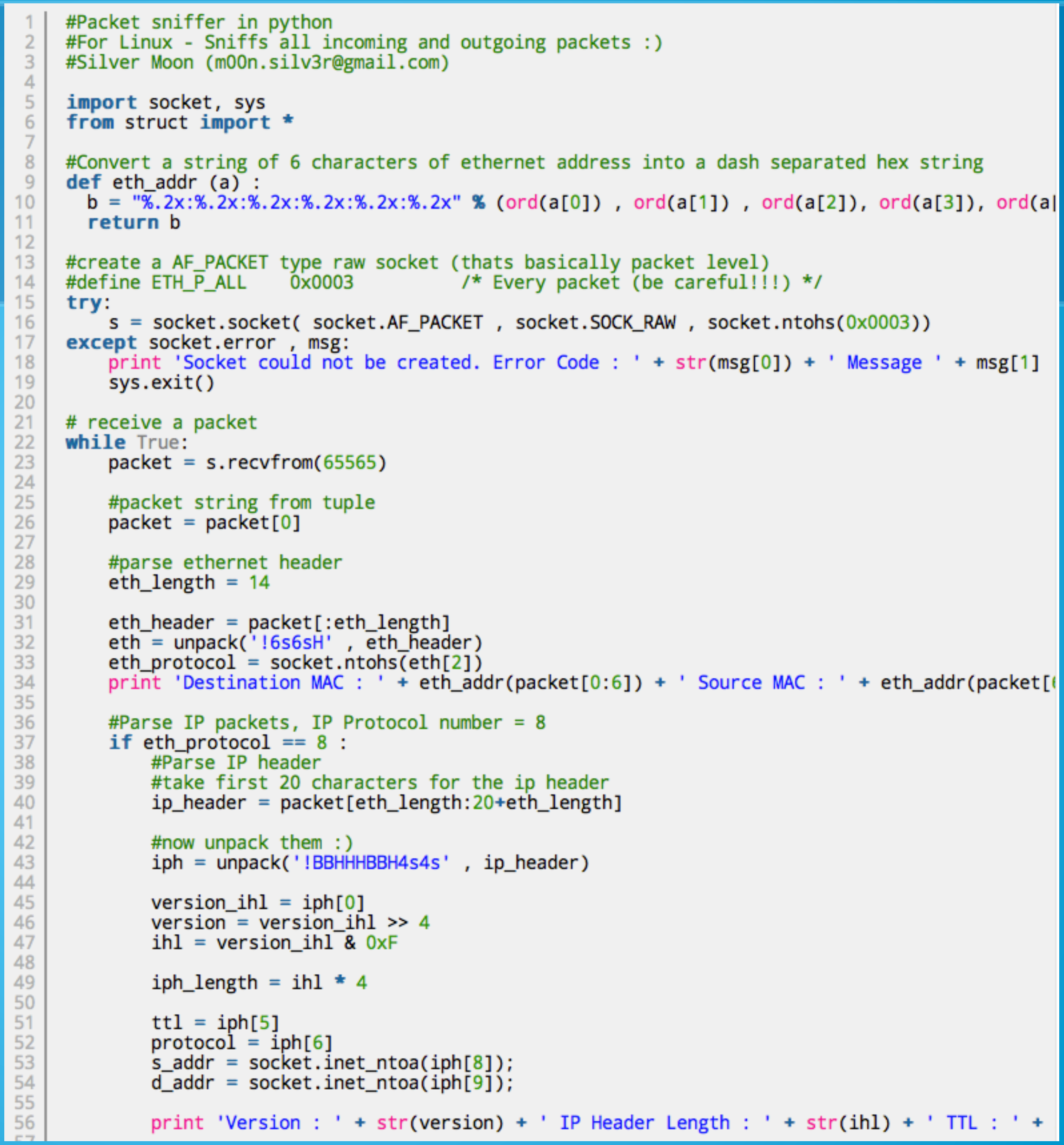
```
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_TCP)
```

needs to be changed to :

```
s = socket.socket( socket.AF_PACKET , socket.SOCK_RAW , socket.ntohs(0x0003))
```

http://www.binarytides.com/python-packet-sniffer-code-linux/

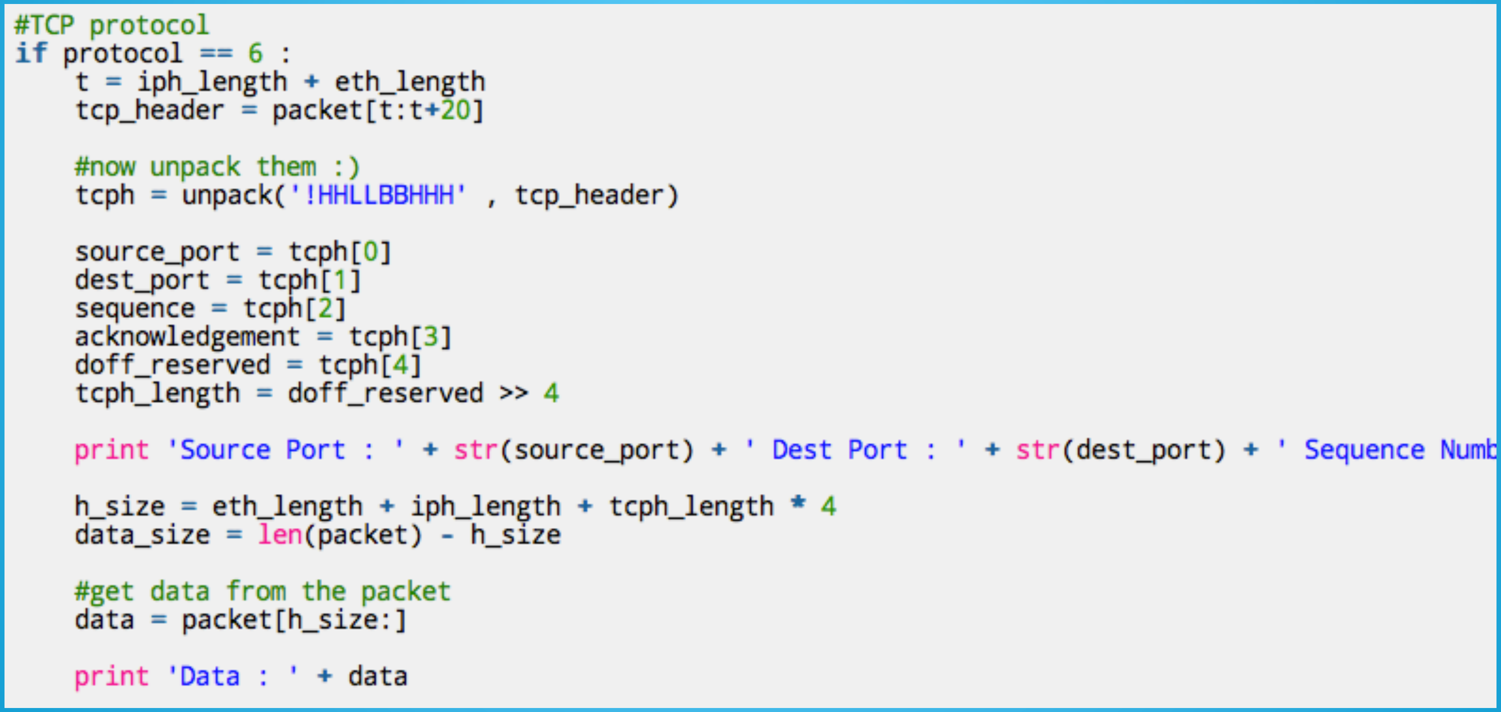
Socket



```
#Packet sniffer in python
#For Linux - Sniffs all incoming and outgoing packets :)
#Silver Moon (m00n.silv3r@gmail.com)

import socket, sys
from struct import *

#Convert a string of 6 characters of ethernet address into a dash separated hex string
def eth_addr (a) :
  b = "%.2x:%.2x:%.2x:%.2x:%.2x:%.2x" % (ord(a[0]) , ord(a[1]) , ord(a[2]), ord(a[3]), ord(a
  return b

#create a AF_PACKET type raw socket (thats basically packet level)
#define ETH_P_ALL    0x0003          /* Every packet (be careful!!!) */
try:
    s = socket.socket( socket.AF_PACKET , socket.SOCK_RAW , socket.ntohs(0x0003))
except socket.error , msg:
    print 'Socket could not be created. Error Code : ' + str(msg[0]) + ' Message ' + msg[1]
    sys.exit()

# receive a packet
while True:
    packet = s.recvfrom(65565)

    #packet string from tuple
    packet = packet[0]

    #parse ethernet header
    eth_length = 14

    eth_header = packet[:eth_length]
    eth = unpack('!6s6sH' , eth_header)
    eth_protocol = socket.ntohs(eth[2])
    print 'Destination MAC : ' + eth_addr(packet[0:6]) + ' Source MAC : ' + eth_addr(packet[

    #Parse IP packets, IP Protocol number = 8
    if eth_protocol == 8 :
        #Parse IP header
        #take first 20 characters for the ip header
        ip_header = packet[eth_length:20+eth_length]

        #now unpack them :)
        iph = unpack('!BBHHHBBH4s4s' , ip_header)

        version_ihl = iph[0]
        version = version_ihl >> 4
        ihl = version_ihl & 0xF

        iph_length = ihl * 4

        ttl = iph[5]
        protocol = iph[6]
        s_addr = socket.inet_ntoa(iph[8]);
        d_addr = socket.inet_ntoa(iph[9]);

        print 'Version : ' + str(version) + ' IP Header Length : ' + str(ihl) + ' TTL : ' +
```

http://www.binarytides.com/python-packet-sniffer-code-linux/

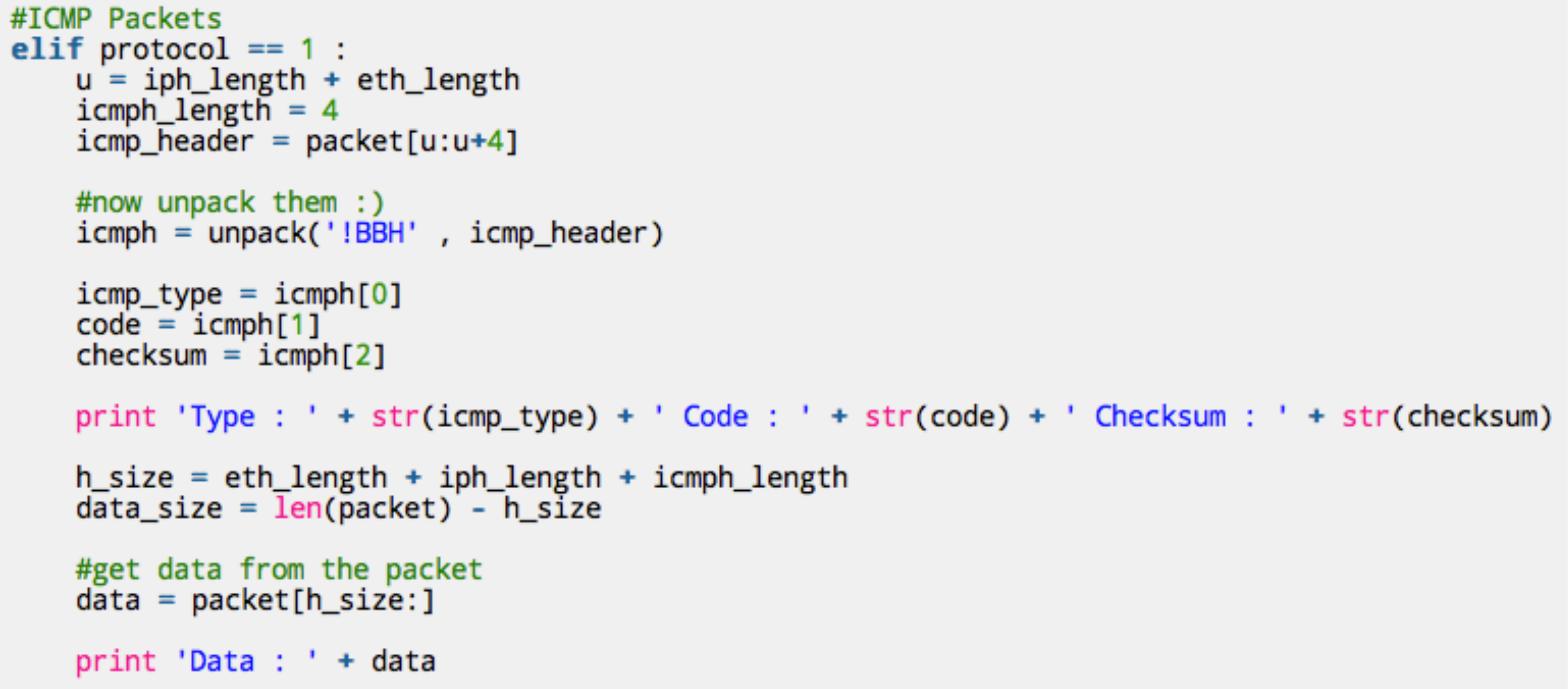
Socket



```
#TCP protocol
if protocol == 6 :
    t = iph_length + eth_length
    tcp_header = packet[t:t+20]

    #now unpack them :)
    tcph = unpack('!HHLLBBHHH' , tcp_header)

    source_port = tcph[0]
    dest_port = tcph[1]
    sequence = tcph[2]
    acknowledgement = tcph[3]
    doff_reserved = tcph[4]
    tcph_length = doff_reserved >> 4

    print 'Source Port : ' + str(source_port) + ' Dest Port : ' + str(dest_port) + ' Sequence Numb

    h_size = eth_length + iph_length + tcph_length * 4
    data_size = len(packet) - h_size

    #get data from the packet
    data = packet[h_size:]

    print 'Data : ' + data
```

http://www.binarytides.com/python-packet-sniffer-code-linux/

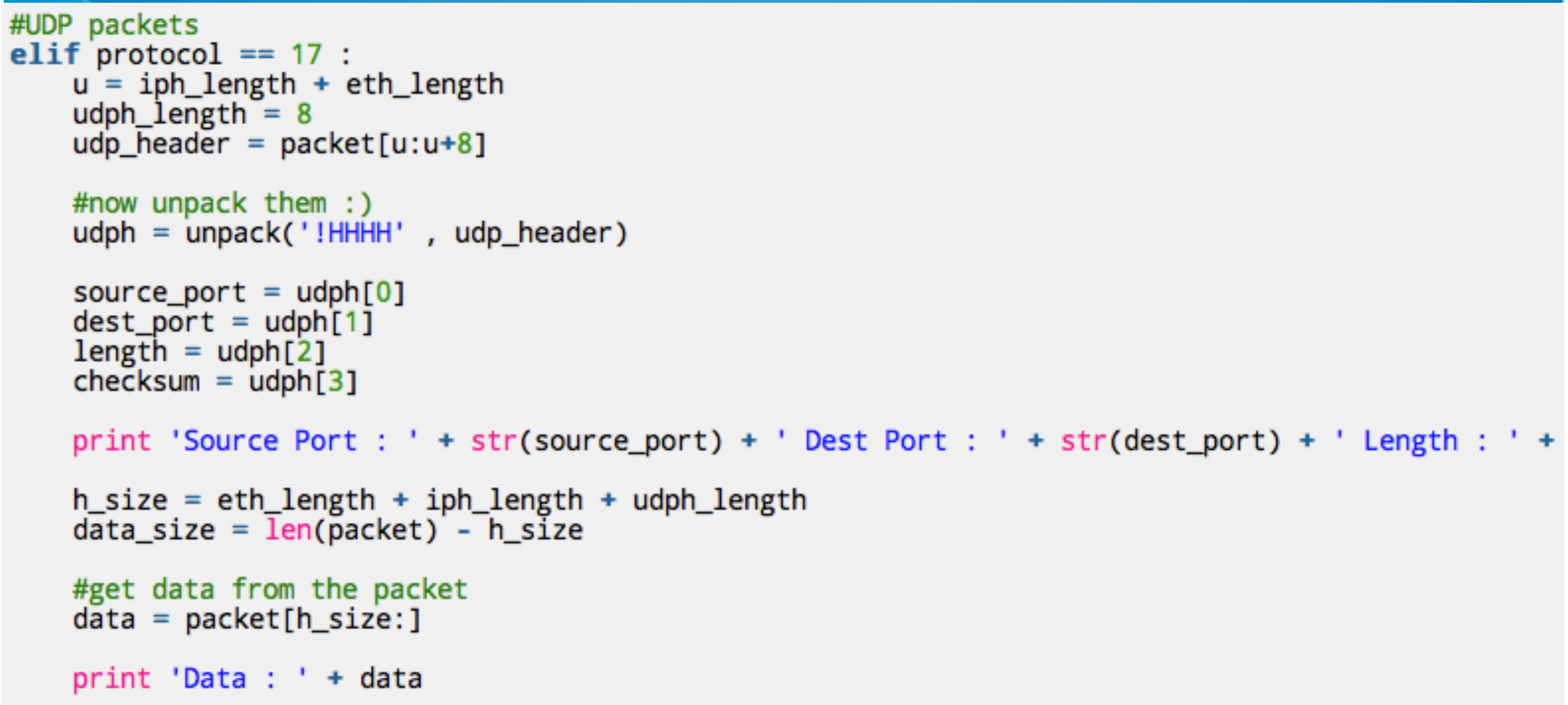
Socket



```
#ICMP Packets
elif protocol == 1 :
    u = iph_length + eth_length
    icmph_length = 4
    icmp_header = packet[u:u+4]

    #now unpack them :)
    icmph = unpack('!BBH' , icmp_header)

    icmp_type = icmph[0]
    code = icmph[1]
    checksum = icmph[2]

    print 'Type : ' + str(icmp_type) + ' Code : ' + str(code) + ' Checksum : ' + str(checksum)

    h_size = eth_length + iph_length + icmph_length
    data_size = len(packet) - h_size

    #get data from the packet
    data = packet[h_size:]

    print 'Data : ' + data
```

http://www.binarytides.com/python-packet-sniffer-code-linux/

Socket

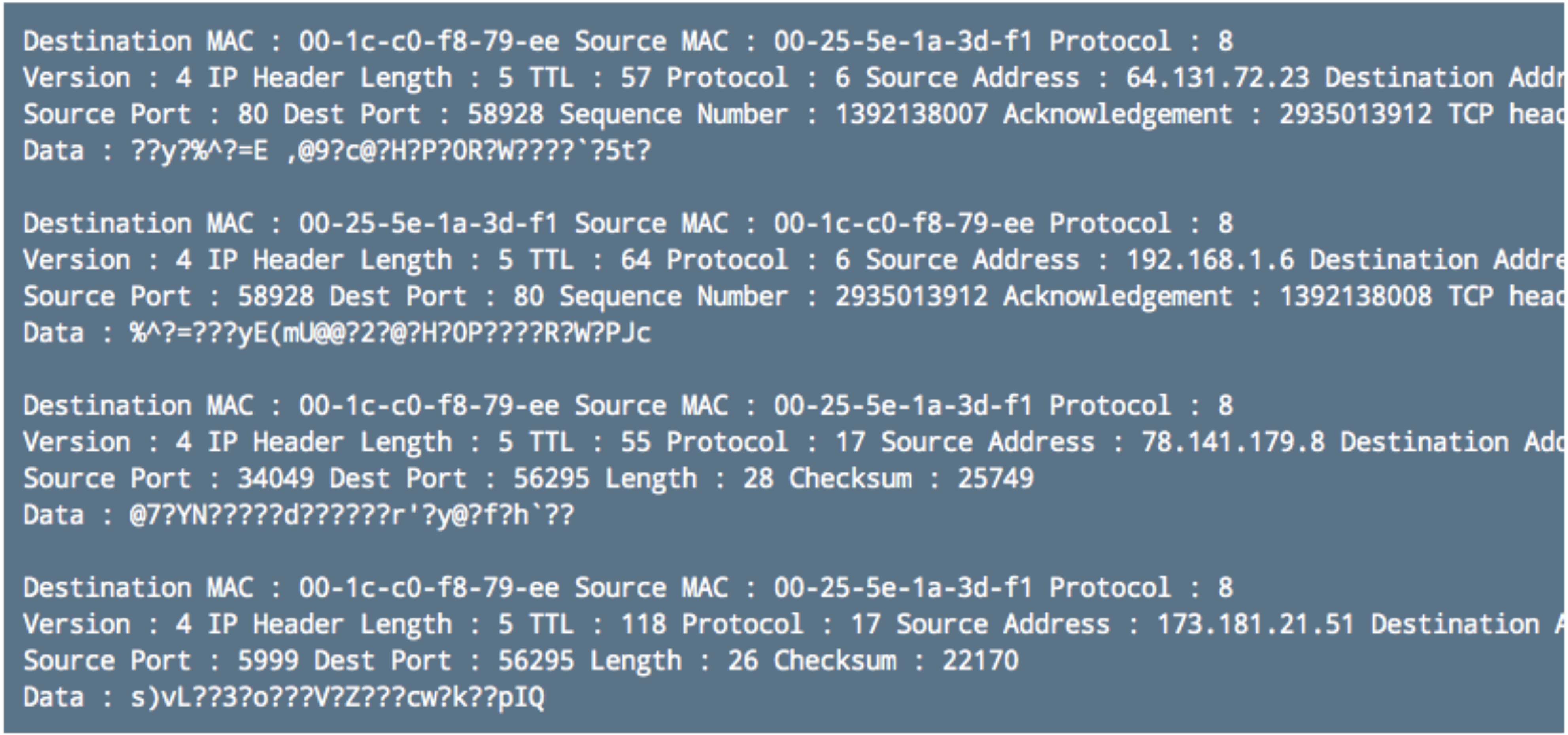


```
#UDP packets
elif protocol == 17 :
    u = iph_length + eth_length
    udph_length = 8
    udp_header = packet[u:u+8]

    #now unpack them :)
    udph = unpack('!HHHH' , udp_header)

    source_port = udph[0]
    dest_port = udph[1]
    length = udph[2]
    checksum = udph[3]

    print 'Source Port : ' + str(source_port) + ' Dest Port : ' + str(dest_port) + ' Length : ' +

    h_size = eth_length + iph_length + udph_length
    data_size = len(packet) - h_size

    #get data from the packet
    data = packet[h_size:]

    print 'Data : ' + data
```

http://www.binarytides.com/python-packet-sniffer-code-linux/

Socket

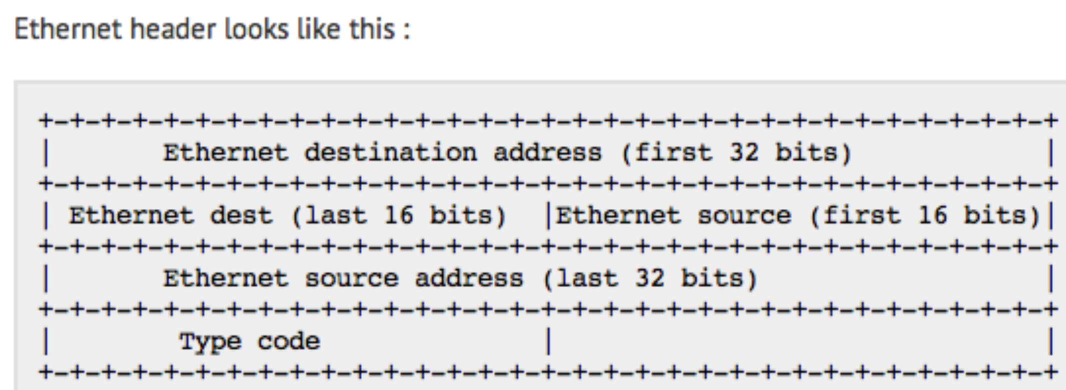
The output should be something like this :



```
Destination MAC : 00-1c-c0-f8-79-ee Source MAC : 00-25-5e-1a-3d-f1 Protocol : 8
Version : 4 IP Header Length : 5 TTL : 57 Protocol : 6 Source Address : 64.131.72.23 Destination Addr
Source Port : 80 Dest Port : 58928 Sequence Number : 1392138007 Acknowledgement : 2935013912 TCP head
Data : ??y?%^?=E ,@9?c@?H?P?0R?W????`?5t?

Destination MAC : 00-25-5e-1a-3d-f1 Source MAC : 00-1c-c0-f8-79-ee Protocol : 8
Version : 4 IP Header Length : 5 TTL : 64 Protocol : 6 Source Address : 192.168.1.6 Destination Addre
Source Port : 58928 Dest Port : 80 Sequence Number : 2935013912 Acknowledgement : 1392138008 TCP head
Data : %^?=???yE(mU@@?2?@?H?0P????R?W?PJc

Destination MAC : 00-1c-c0-f8-79-ee Source MAC : 00-25-5e-1a-3d-f1 Protocol : 8
Version : 4 IP Header Length : 5 TTL : 55 Protocol : 17 Source Address : 78.141.179.8 Destination Add
Source Port : 34049 Dest Port : 56295 Length : 28 Checksum : 25749
Data : @7?YN?????d??????r'?y@?f?h`??

Destination MAC : 00-1c-c0-f8-79-ee Source MAC : 00-25-5e-1a-3d-f1 Protocol : 8
Version : 4 IP Header Length : 5 TTL : 118 Protocol : 17 Source Address : 173.181.21.51 Destination A
Source Port : 5999 Dest Port : 56295 Length : 26 Checksum : 22170
Data : s)vL??3?o???V?Z???cw?k??pIQ
```

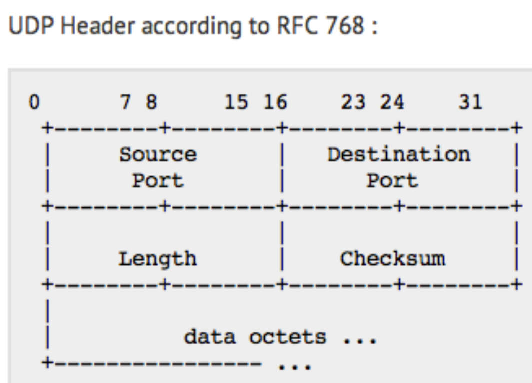
http://www.binarytides.com/python-packet-sniffer-code-linux/

Socket



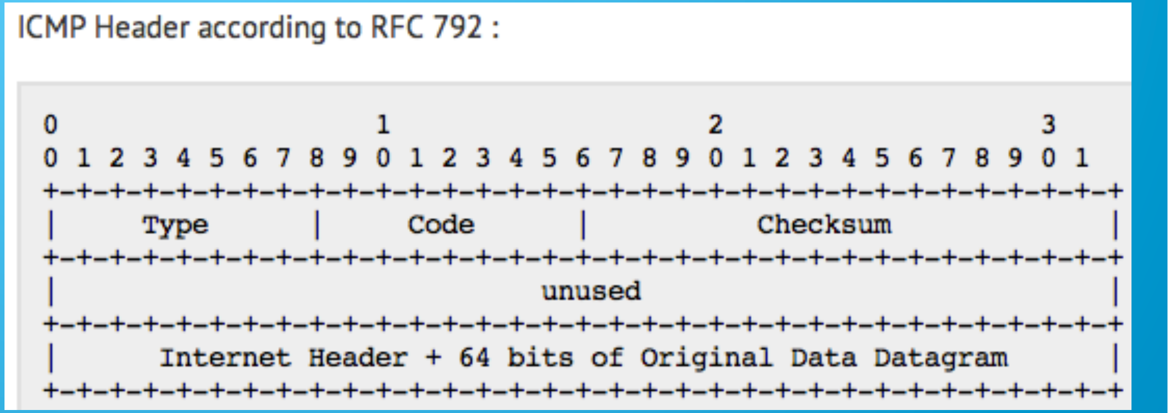
Ethernet header looks like this :

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Ethernet destination address (first 32 bits)           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Ethernet dest (last 16 bits)  |Ethernet source (first 16 bits)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Ethernet source address (last 32 bits)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Type code              |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```



UDP Header according to RFC 768 :

```
 0      7 8     15 16    23 24    31
 +--------+--------+--------+--------+
 |     Source      |   Destination   |
 |      Port       |      Port       |
 +--------+--------+--------+--------+
 |                 |                 |
 |     Length      |    Checksum     |
 +--------+--------+--------+--------+
 |
 |          data octets ...
 +---------------- ...
```



ICMP Header according to RFC 792 :

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |     Code      |          Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             unused                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Internet Header + 64 bits of Original Data Datagram      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

http://www.binarytides.com/python-packet-sniffer-code-linux/