Lecture 7: Introduction of TCP/IP and Socket API

(Chapter 1, 2, 3 of 2nd textbook)

(10/21/2009)

Lecture Outline

```
1. The client/server application model
2. Overview of TCP/IP protocol suite
3. IP: the heart of TCP/IP suite
4. ARP, RARP, and local address resolution
5. UDP -- User Datagram Protocol
6. TCP -- Transmission Control Protocol
7. Concurrent servers and port numbers
8. Buffer sizes and limitations
9.  Standard (well known) Internet services
10. Similarities and differences between file I/O and network I/O
11. Sockets and socket structures
12. Utility functions
```

1. The Client/server application model

(1) Client/server application model

a. Basic structure of a client-server model: Fig.1.1,p.3.

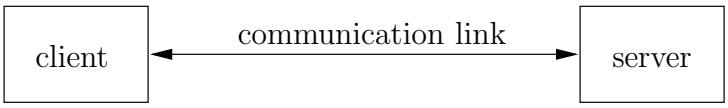


Figure 51: Network application: client and server. (Fig.1.1,p.3)

b. The communication protocol between the client and server can be TCP/IP or others (such as pipes if client and sever are on the same host).

c. A server can simultaneously serve multiple clients (Fig.1.2,p.4)

(2) Application program interfaces (APIs) to the communication protocols.

a. The API is the interface available to the programmers. Availability of APIs depends on both the OS and programming languages.

b. Two most popular APIs for UNIX systems: System V Transport Layer Interface (TLI) and Berkeley sockets. Both were developed for the C language.

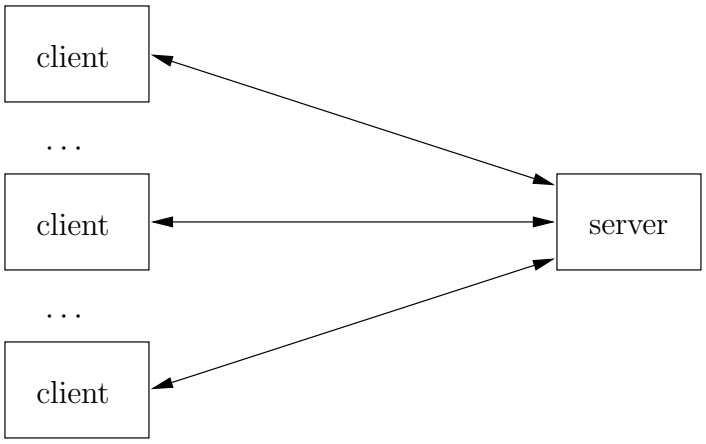


Figure 52: Server handling multiple clients at the same time. (Fig.1.2,p.4)

(3) Client/server application model with TCP/IP (Fig.1.3,p.4, Fig.1.4,p.5)

a. Client and server on the same Ethernet

b. Client and server on different ethernets through WAN

(4) A first TCP/IP networking example

a. A simple daytime client (p.6, Fig.1.5)

* The *sockaddr_in* structure
* The *socket* function
* The *bzero* function
* The *htons* function
* The *inet_pton* function
* The *connect* function

```
1  #include "unp.h"

2  int
3  main(int argc, char **argv)
4  {
5      int    sockfd;
6      char   recvline[MAXLINE + 1];
7      struct sockaddr_in servaddr;

8      if (argc != 2)
9         err_quit("usage: a.out <IPaddress>");
```

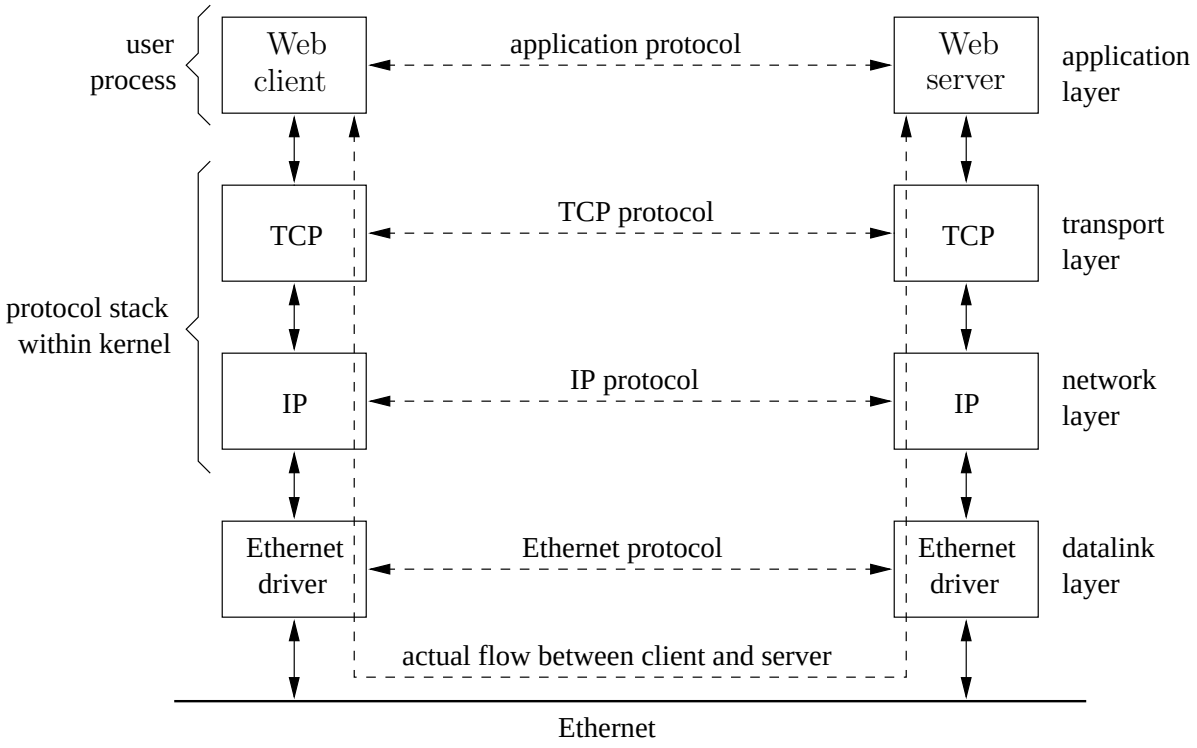


Figure 53: Client and server on the same Ethernet communicating using TCP (Fig.1.3,p.4)

```
10      if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
11          err_sys("socket error");

12      bzero(&servaddr, sizeof(servaddr));
13      servaddr.sin_family = AF_INET;
14      servaddr.sin_port = htons(13);       /* daytime server */
15      if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
16          err_quit("inet_pton error for %s", argv[1]);

17      if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
18          err_sys("connect error");

19      while ((n = read(sockfd, recvline, MAXLINE)) > 0) {
20          recvline[n] = 0;    /* null terminate */
21          if (fputs(recvline, stdout) == EOF)
22             err_sys("fputs error");
23      }
```

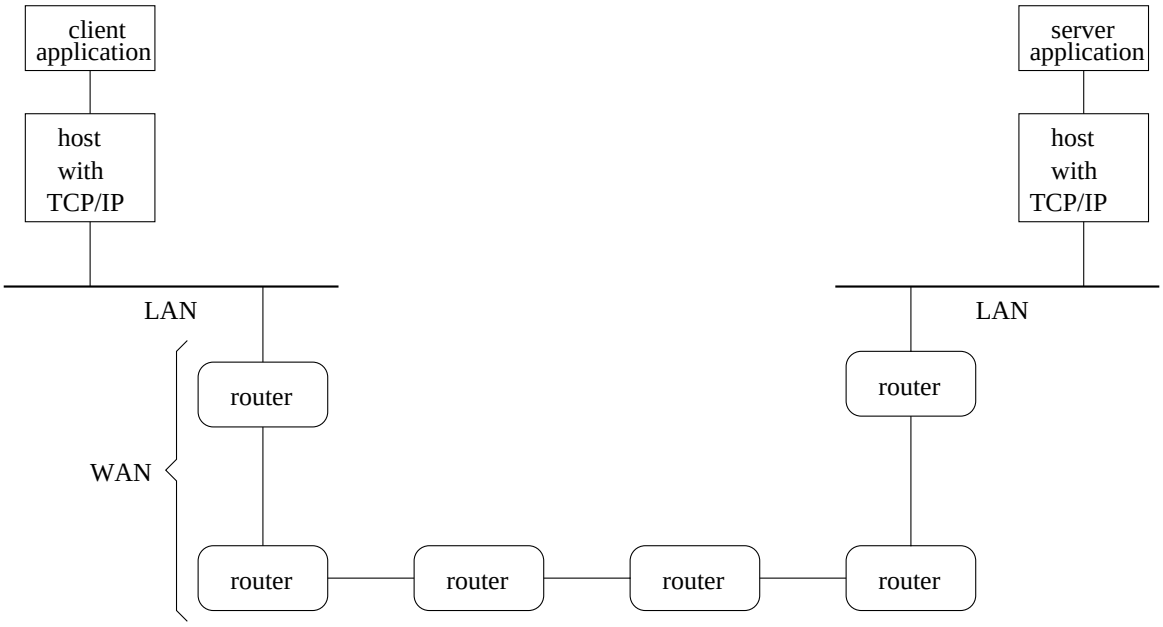


Figure 54: Client and server on different LANS connected through a WAN (Fig.1.4,p.5)

```
24      if (n < 0)
25          err_sys("read error");

26      exit(0);
27  }
```

Figure 1.5 TCP daytime client

b. Protocol dependency:

(a) Change the protocol from IPv4 to IPv6 in Fig.1.5 (p.10,Fig.1.6)

(b) An IP version independent daytime client is shown in Fig.11.7.

```
1   #include "unp.h"

2   int
3   main(int argc, char **argv)
4   {
5       int    sockfd;
6       char   recvline[MAXLINE + 1];
7       struct sockaddr_in6 servaddr;

8       if (argc != 2)
9           err_quit("usage: a.out <IPaddress>");
```

```
10      if ((sockfd = socket(AF_INET6, SOCK_STREAM, 0) < 0))
11          err_sys("socket error");

12      bzero(&servaddr, sizeof(servaddr));
13      servaddr.sin6_family = AF_INET6;
14      servaddr.sin6_port = htons(13);       /* daytime server */
15      if (inet_pton(AF_INET6, argv[1], &servaddr.sin6_addr) <= 0)
16          err_quit("inet_pton error for %s", argv[1]);

17      if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
18          err_sys("connect error");

19     while ((n = read(sockfd, recvline, MAXLINE)) > 0) {
20          recvline[n] = 0;    /* null terminate */
21          if (fputs(recvline, stdout) == EOF)
22              err_sys("fputs error");
23      }

24      if (n < 0)
25          err_sys("read error");

26      exit(0);
27  }
```

Figure 1.6 Version of Figure 1.5 for IP version 6

c. Error handling with wrapper functions: Fig.1.7, p.11. Simplied coding and reduced typing error for each function (from lib/wrapsock.c file).

```
172 int
173 Socket(int family, int type, int protocol)
174 {
175     int  n;

176     if ((n = socket(family, type, protocol)) < 0)
177         err_sys("socket error");
178     return (n);
179 }
```

Figure 1.7 Our wrapper function for the *socket* function.

d. UNIX *errno* variable

(a) A UNIX function will normally return a value 0 if it returns successfully and a value -1 if returns with errors.

(b) The variable *errno* is a global variable. When a function call returns with error, this variable is set to *positive* number indicating the type of errors.

(c) The types of errors each positive number returned in errno are normally defined in system header file <sys/errno.h>.

(d) A multithread program will have exercise caution if all global variables are shared among threads.

e. The daytime server: p.14, Fig.1.9

```
1  #include "unp.h"
2  #include <time.h>

3  int
4  main(int argc, char **argv)
5  {
6      int              listenfd, connfd;
7      struct           sockaddr_in servaddr;
8      char             buff[MAXLINE];
9      time_t           ticks;

10     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
14     servaddr.sin_port = htons(13);       /* daytime server */

15     Bind(listenfd, (SA *) &servaddr, sizeof (servaddr));
16     Listen (listenfd, LISTENQ);

17     for ( ; ; ) {
18         connfd = Accept(listenfd, (SA *) NULL, NULL);

19         ticks = time(NULL);
20         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
21         write(connfd, buff, strlen(buff));

22         close(connfd);
```

```
23      }
24 }
```

Figure 1.9 TCP daytime server.

* The constant INADDR_ANY.
* The *htonl* function
* The *bind* function
* The *listen* function
* The *accept* function
* Terminating a connection

(5) Relationship between the OSI model and TCP/IP protocol suite: p.19, Fig.1.14

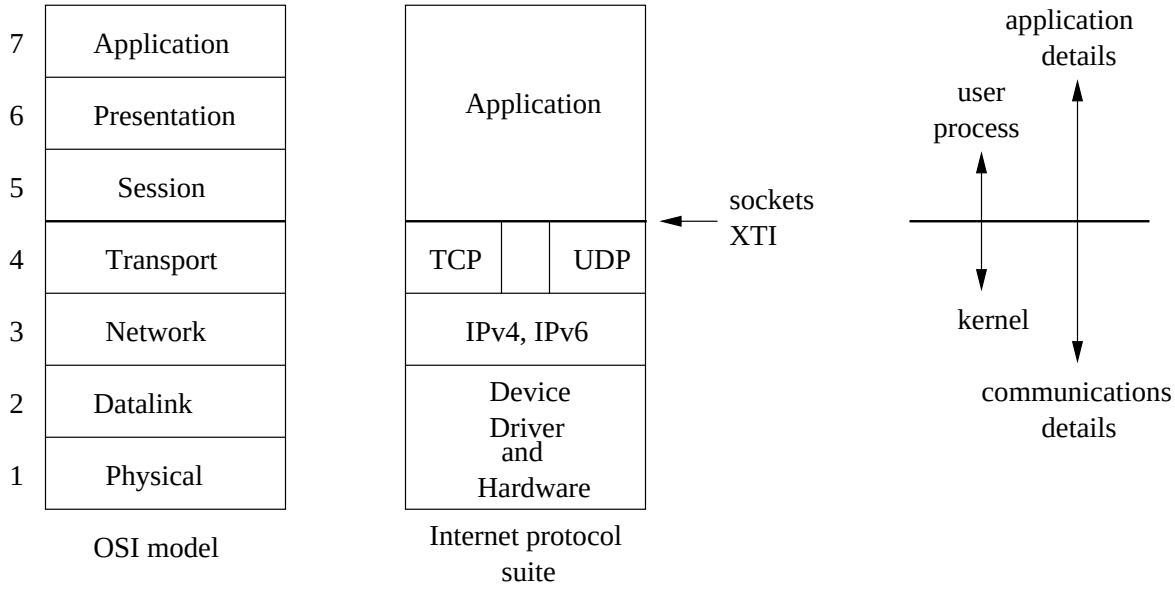


Figure 55: Layers in OSI model and IP suite (Fig.1.14,p.18)

(6) BSD networking history: Fig.1.15, p.21

(7) Test networks and hosts: p.22, Fig.1.16

- The example network environment: Fig.1.16
- The *traceroute* command
- The *netstat* command
- The *ifconfig* command
- The *ping* command

(8) UNIX standards

- POSIX
- The Open Group
- Internet Engineering Task Group

(9) 64-bit architecture

2. Overview of TCP/IP protocol suite (Chapter 2, 2nd textbook)

(1) Overview – the big picture about TCP/IP (Fig. 2.1 p.32)
There are more members of this family than just TCP and IP

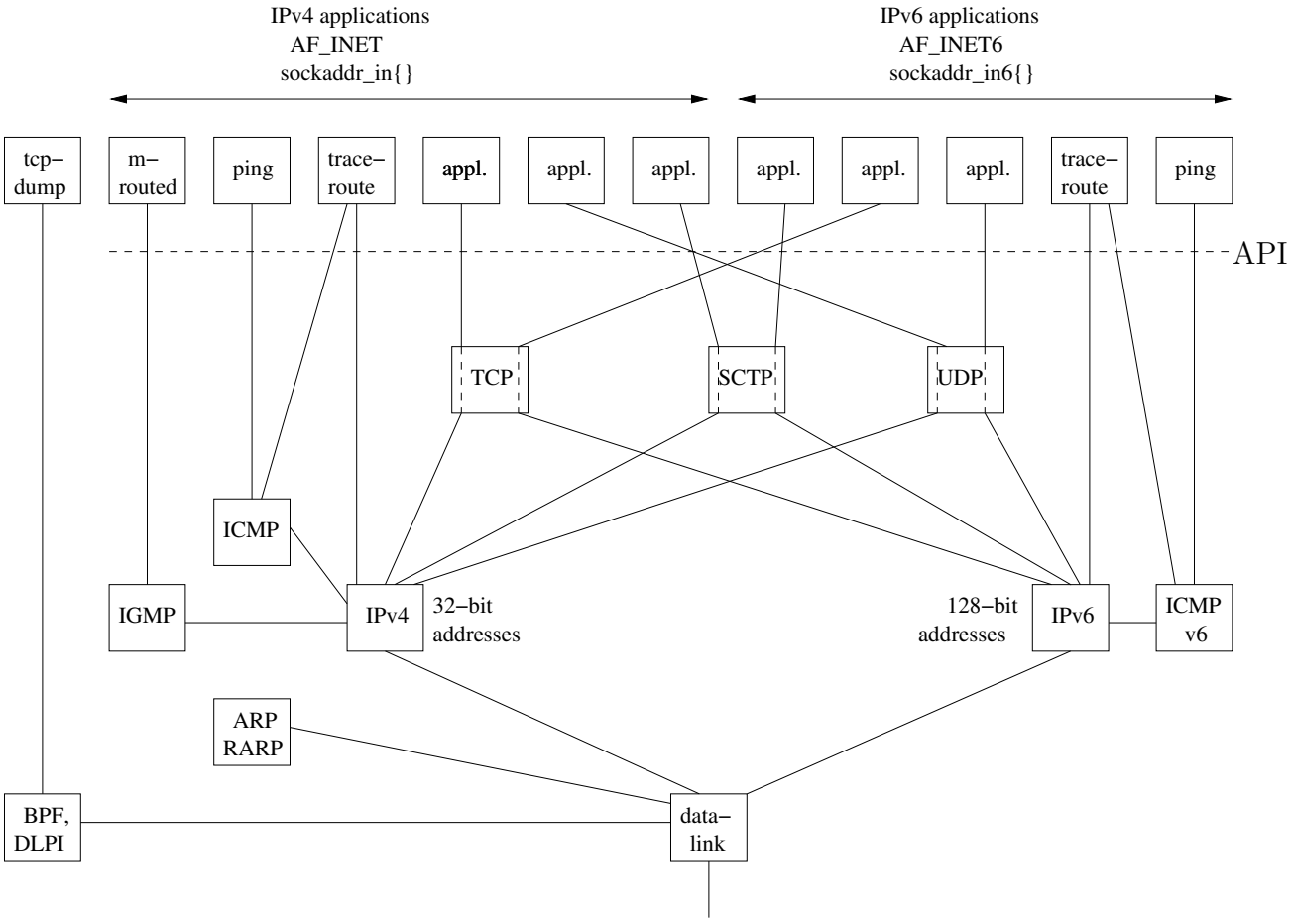


Figure 56: Overview of TCP/IP protocols (Fig.2.1,p.32)

IPv4 *Internet protocol version 4*, commonly called IP, has been the protocol that has been at the center of TCP/IP protocol suite since 1980s. It uses 32-bit addresses

(Appendix A.4) and provides the packet delivery service for TCP, UDP, SCTP, ICMP, and IGMP. User processes normally do not need to get involved into the IP layer.

IPv6 *Internet protocol version 6*, was designed in the mid-1990s. It uses 128-bit addresses (Appendix A.5) to accomodate the explosive expansion of the Internet in 1990s. It provides the packet delivery service for TCP, UDP, SCTP, and ICMPv6.

TCP *Transmission control protocol*, is a connection oriented protocol that provides reliable, full-duplex, byte stream for a user process. TCP uses IP, so the entire Internet protocol suite is ofter referred as TCP/IP protocol family.

UDP *User datagram protocol*, is a connectionless protocol. There is no guarantee that user UDP datagrams will reach their intended destination.

SCTP *Stream control transmission protocol*, is a connection-oriented protocol that provides a reliable full-duplex association. The term *association* means that when a connection in SCTP is made it may involve multihomed IP addresses of either or both end of two sides of the connection. SCTP provides a message service, which maintains record boundaries. Similar to TCP and UDP, SCTP can use either IPv4 or IPv6, but it can also use both IPv4 and IPv6 simultaneously on the same association.

ICMP *Internet control message protocol*, handles errors and control information between gateways and hosts. While ICMP messages are transmitted using IP datagrams, they are normally generated by networking software, not user processes.

ARP *Address resolution protocol*, is the protocol that maps an Internet address into an hardware address. These protocols and the next (RARP) are used on some of the networks.

RARP *Reverse address resolution protocol*, is the protocol that maps a hardware address into an Internet address.

IGMP *Internet group management protocol*, is used by hosts and routers to support multicasting.

BPF *BSD Packet Filter*, is an interface that provides access to the datalink for a process. Normally supported on BSD-derived kernels.

DLPI *Datalink Provider Interface*, is an interface that provides access to the datalink for a process on UNIX systems derived from SVR4.

(2) Other protocols not studied in the book

a. GGP (Gateway-to-gateway protocol)

b. VMTP (Versatile message transaction protocol)

(3) All Internet protocols are defined by RFCs (*request for comments*) and are formally specified there.

3. IP: the heart of TCP/IP suite

(1) IPv4 datagrams

a. The IP provides a connectionless and unreliable delivery system. It is connectionless because it considers each datagram independent of others. All association between datagrams must be provided by upper layers.

b. Each IP datagram contains the source address and the destination address so that they can be deliveried independently.

c. The IP layer is unreliable because it does not guarantee that each datagram will be delivered at all or delivered correctly. Reliability must be provided by the upper layers.

d. The IP layer computes and verifies a checksum that covers its own 20-byte header (containing source and destination address). This is the only field that it needs to examine and process. An datagram with an incorrect IP header is simply discarded.

e. IP layer handles routing and fragmentation (divide large IP datagrams into smaller IP packets) through the Internet. Each IP packet carries the source and destination addresses found from its original IP datagram. The fragments are reassembled into an IP datagram only when they reach their final destination. An IP datagram is discarded if any of its IP packets are lost or discarded.

f. The IP layer provides some elementary form of flow control. When it is flooded it simply discards incoming packets and sends an ICMP message to ask the original hosts or gateways to slow down.

(2) IPv4 addresses (Appendix A.4)

a. An IPv4 internet address occupies 32 bits, and encode both the network ID and host ID. The host ID is relative to the network ID. Every host on a TCP/IP internet must have a unique 32 bit address.

b. TCP/IP addresses on the Internet are assigned by a central authority – the Network Information Center (NIC, previously located at SRI International).

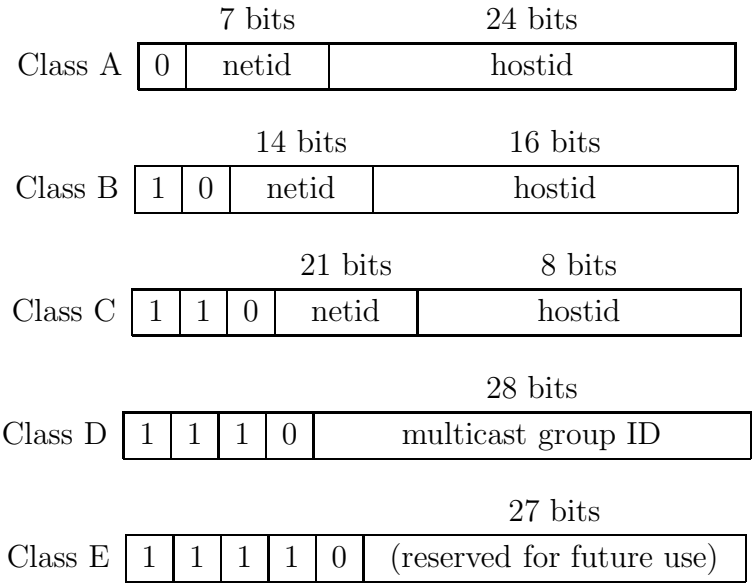
* Now the Internet domain name registrations are handled by individual countries.
* In US, check the URL:

http://www.internic.net

and

http://www.nic.net

c. A 32-bit Internet address has one of the four formats shown as follows:



(3) IPv4 address classes

a. Address ranges:

Usage	Class	Range
Unicast	A	**0**.0.0.0 to **127**.255.255.255
Unicast	B	**128**.0.0.0 to **191**.255.255.255
Unicast	C	**192**.0.0.0 to **223**.255.255.255
Multicast	D	**224**.0.0.0 to **239**.255.255.255
Experimental	E	**240**.0.0.0 to **255**.255.255.255

Figure A.3 Ranges for the five different classes of IPv4 addresses.

b. Class A addresses are used for those networks that have a lot of hosts on a single network.

c. Class C addresses allow more hosts but fewer hosts per network.

d. The NIC only assigns the type of addresses (A, B, or C) and the network ID. The requesting network then has responsibility to assign individual addresses within that network.

e. Internet addresses are usually written as four decimal numbers, separated by decimal point. Each decimal digit encodes one byte of the 32-bit Internet address. For example, the 32-bit hexadecimal value 0x0102FF04 is written as 1.2.255.4, which represents a Class A address with network ID 1 and a host ID 2.255.4. A sample Class B address is 128.3.0.5, and a sample Class C address is 192.43.236.6

f. Every IP datagram contains two 32-bit addresses: source and destination, in every 20-byte IP header. Gateways can use the network ID for routing.

g. Each Internet address specifies a unique host, but each host may not have a unique address – some are multihomed hosts.

(4) Subnet addresses

a. Each org. with an Internet address of any class can subdivide the available host space in any way it wants to provide subnetworks.

b. For instance, if with a class B address, there is 16 bits allocated for the host ID. If an org. wants to assign IDs to its 150 hosts, that are in turn organized into 10 physical networks, there are two different ways:

   (a) Allocate hosts ID from 1 to 150, ignoring the physical network structure. This requires that all the gateway systems among the 150 hosts know where each individual host is located for routing.

   (a) Allocate some of the high-order bits from the host ID (say high 8-bit) for the network ID within your subnetwork. The remaining 8 bits of the Class B host ID is then used to identify the individual hosts. Adding new hosts in any existing internal network does not require any changes to the internal gateways

c. Subnetting provides another level to the address hierarchy:

   * Network ID (assigned to site). The bounday between the network ID and subnet ID is fixed by the prefix length of the assigned network address. This prefix length is normally assigned by the organization's ISP (Internet service provider).
   * Subnet ID (chosen by site).
     · The boundary between the subnet ID and host ID is chosen by the organization.

- All the hosts on a given subnet share a common *subnet mask*, which specifies the boundary between the subnet ID and the host ID. For example a subnet mask 255.255.255.0 means that the network and subnet IDs together have (the first) 24 bits, and the host ID use (the last) 8 bits.

* Host ID (chosen by site)

d. Example. Consider an organization that is assigned 192.168.42.0/24 by its ISP.

(a) The notion of 192.168.42.0/24 means that the organization's network ID is the first 24 bit of the IP 192.168.42.0

(b) The organization can then further subnet its last 8-bit portion of the assigned IP addresses into a 3-bit subnet ID, 5 bits are left for the host IDs within the organization. This is illustrated by Fig.A.4, p.975 (Fig.A.5 Subnet list for 3-bit subnet ID and 5-bit host ID):

Subnet	Prefix
0	192.168.42.0/27
1	192.168.42.32/27
2	192.168.42.64/27
3	192.168.42.96/27
4	192.168.42.128/27
5	192.168.42.160/27
6	192.168.42.192/27
7	192.168.42.224/27

Figure A.5 Subnet list for 3-bit subnet ID and 5-bit host ID.

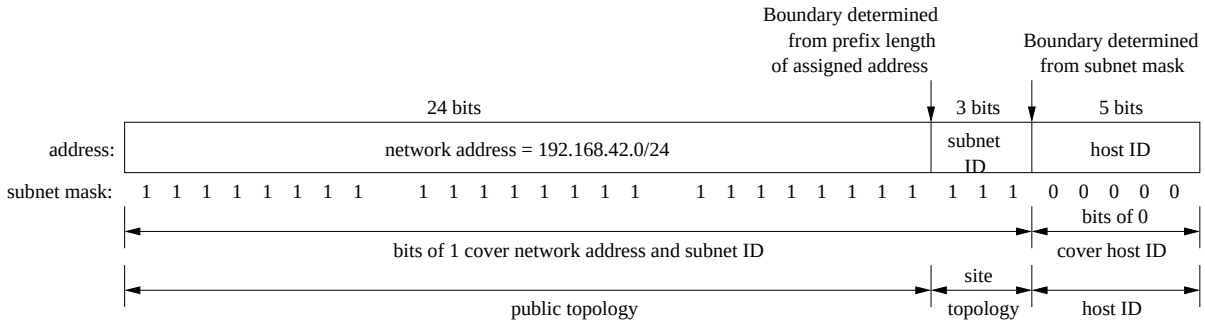


Figure 57: 24-bit network address with 3-bit subnet ID and 5-bit host ID (Fig.A.4,p.875)

(5) Loopback addresses

a. The address 127.0.0.1 is the special loop back address.

b. Any data sent to this special address will be treated as input to IP, and the data will not even leave the sending DTE.

(6) IPv4 headers

a. IPv4 header: p. 870, Fig.A.1

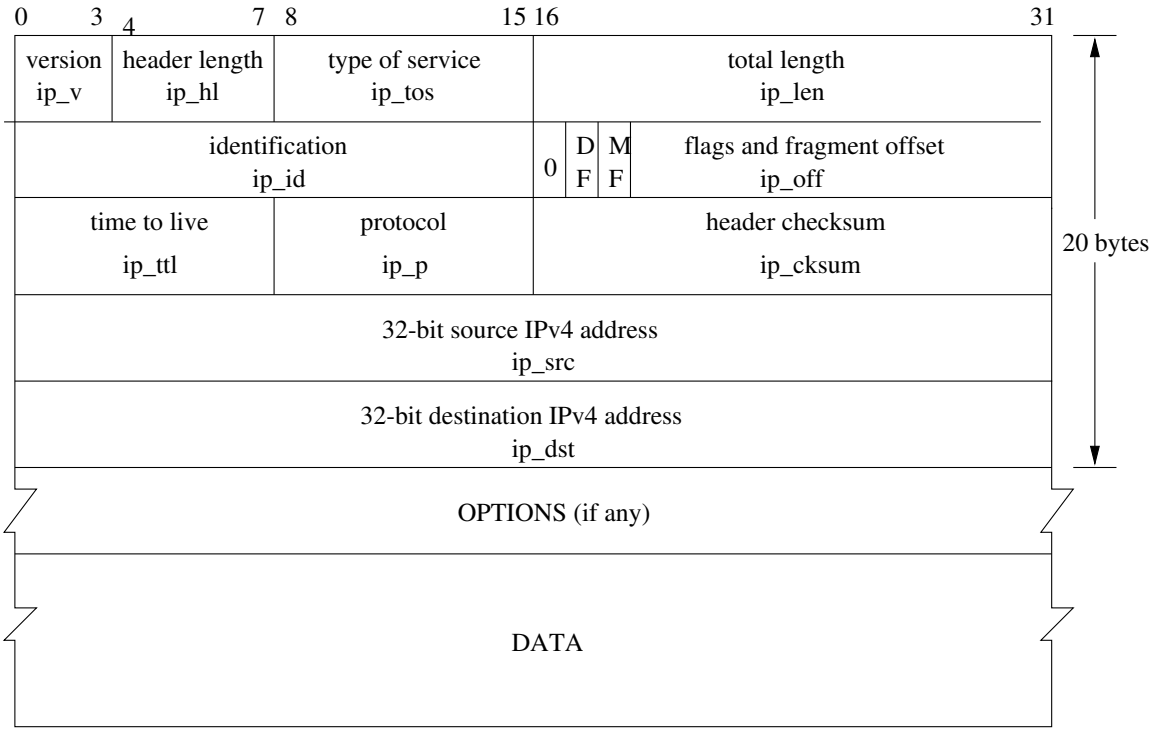


Figure 58: Format of IPv4 Header and Options

(a) Can be up to 60 bytes.

(b) A 4-bit *version* field, containing value 4 (since the early 1980s).

(c) A 4-bit *header length* field. This value records the length of the header in the unit of 32-bit words. Since it is 4 bit, the maximum values is $15 * 4 = 60$ bytes. Among these, the fixed portion of the header occupies 20 bytes, and the option part can use up to 40 bytes.

(d) An 8-bit *type of service* field, which consists of a 3-bit *precedence* field (currently ignored), a 4-bit type-of-service field, and an unused bit that must be equal to 0. The *IP_TOS* option can be used to set this field.

(e) A 16-bit *total length* field, which contains total length in bytes of the IP datagram, including the IP header. This field is required because padding may be used (in case of Ethernet, for instance)

(f) A 16-bit *identification* field, which is set to different value for each IP datagram, and is used with fragmentation and reassembly (ref. Sect.2.9).

(g) A 1-bit *DF* (*don't fragment*) field and a 1-bit *MF* (*more fragments*) field.

(h) A 13-bit *fragment offset* field (also used with fragmentation and reassembly).

(i) An 8-bit *time-to-live* field (TTL), which limits the number of hops that an IP datagram can live on the internet. The sender sets it to certain non-negative integer and each intermediate router will decrease it by one. If this field has zero value after decrementation, a router will discard it.

(j) An 8-bit *protocol* field, which specifies the type of data contained in the IP datagram. Typical value is 1 (ICMPv4), 2 (IGMPv4), 6 (TCP), and 17 (UDP).

(k) A 16-bit *header checksum* field, which is calculated over just the IP header.

(l) A 32-bit IPv4 *source address* field and a 32-bit IPv4 *destination address* field.

b. IPv6 header: p. 872, Fig.A.2

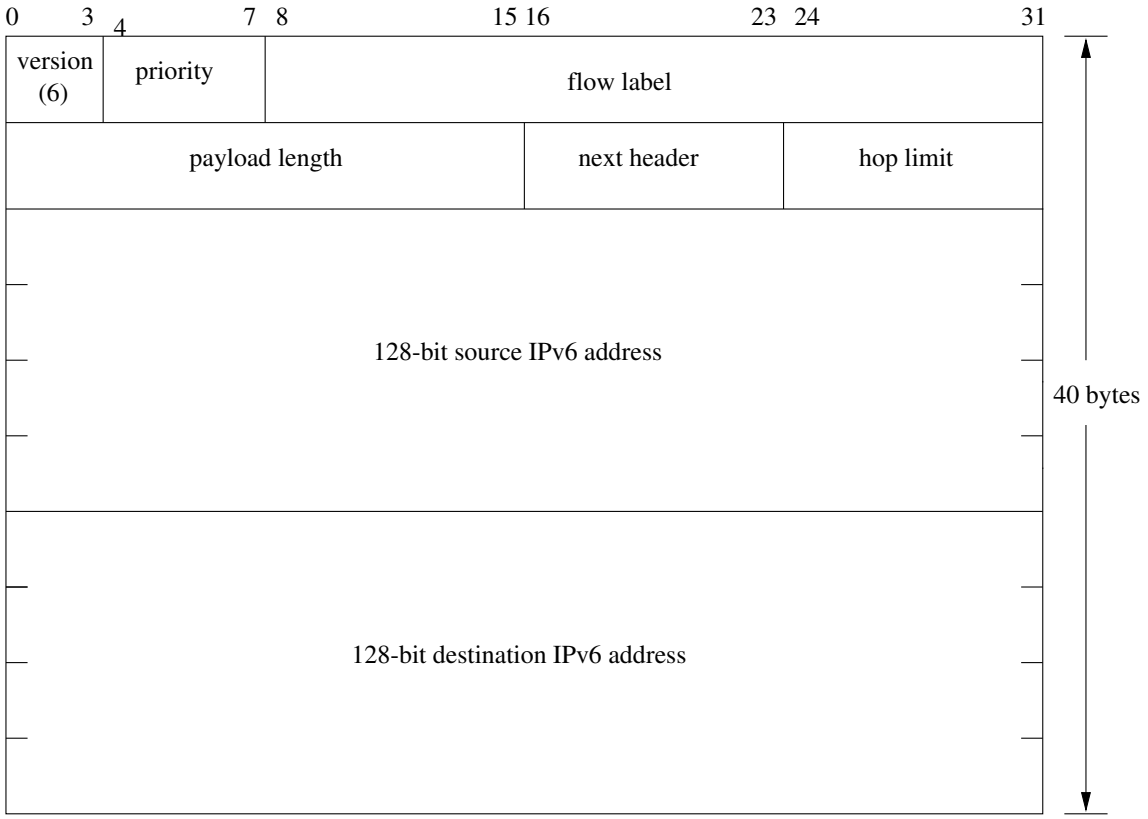


Figure 59: Format of IPv6 Header

4. ARP, RARP, and local address resolution

(1) Address resolution
Each Internet host which is also in a local Ethernet has two addresses: the Internet address and a unique Ethernet address assigned by the manufacture. Two problems:

a. *Address resolution*: how does the IP layer determine the Ethernet address of a remote host?

b. *Reverse address resolution*: When a diskless workstation is rebooted, it can get Ethernet address from its interface hardware. How does the IP layer determine its Internet address?

(2) The ARP and RARP protocols

a. The ARP allows a host to broadcast a special on the Ethernet that asks the host with a special Internet address to respond with its Ethernet address. Once the requesting host receives the response, it can maintain the mapping between the Internet addresses and Ethernet addresses.

b. The RARP protocol is for diskless workstations. One or more systems on the LAN are RARP servers and contain the 32-bit Internet address and its corresponding 48-bit Ethernet address. When a workstation is initialized, it gets its Internet address from the server by broadcasting its Ethernet address.

5. UDP – User Datagram Protocol

(1) Overview (p.34)

(2) Services

a. IP layer provides an unreliable, connectionless delivery service to TCP and UDP.

b. UDP only provides two more services than IP:

* port numbers; and
* an optional checksum to verify the contents of a datagram.

(3) UDP header (from TCP/IP Illustrated Volume 1, Stevens).
Compared with TCP header, it is much simpler.

6. TCP – Transmission Control Protocol

(1) Overview (p.35,36). IP layer provides an unreliable, connectionless delivery service to TCP and UDP.

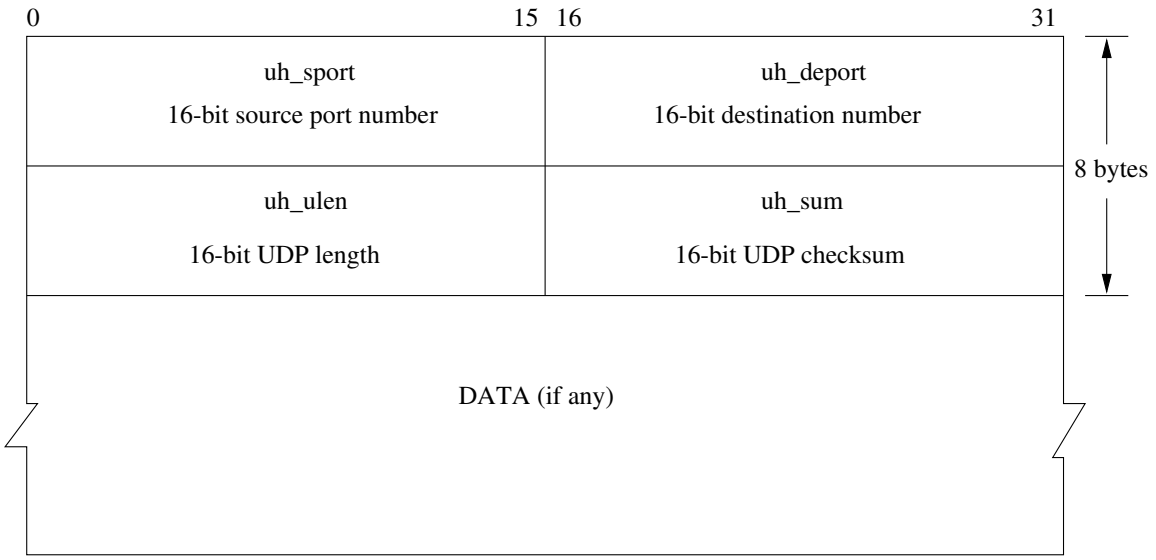


Figure 60: UDP Header and Optional Data

(2) TCP must provide: sequencing, error control, acknowledgment, time-out, resequencing (retransmitting).

a. Reliability: acknowledgment required

b. RTT: used to estimate timeout.

c. Sequencing: sequence numbers are used for each data segment (typically 1024 bytes).

* The first segment will contain sequence numbers 1-1024
* The second segment will contain sequence numbers 1025-2048, and so on.

d. Flow control: TCP always tells the peer how many bytes of data it is willing to receive by adverting its window size.

* The window size is equal to the amount room currently available in the receiver's buffer. Therefore it is guaranteed that the sender will not overflow the receiver.
* The window changes dynamically. It can reach size 0, at which instance the receiver has to wait for the application to read data from the buffer before it can receive further data.

e. TCP communication is full-duplex. Hence TCP must keep track of state information (sequence numbers and window sizes) on both directions: sending and receiving.

(3) TCP connection establishment and termination (p.37-44)

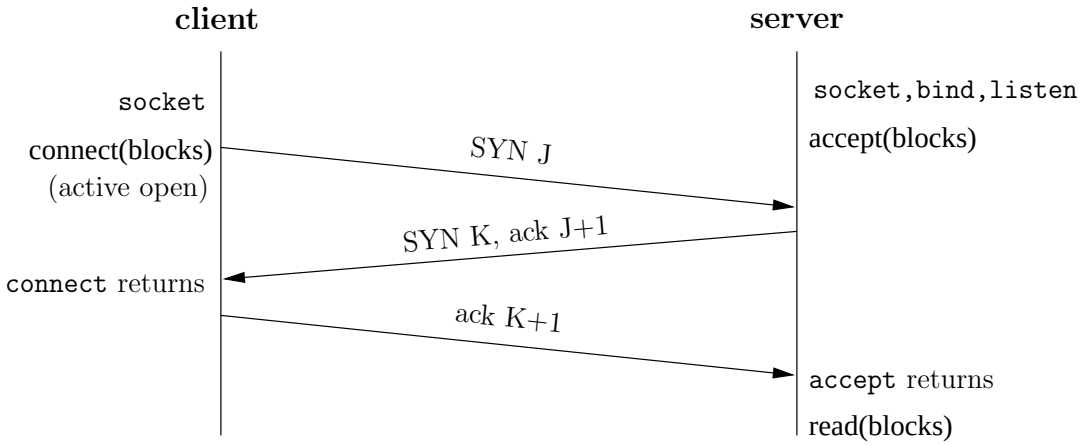


Figure 61: TCP three-way handshake (Fig.2.2,p.37)

a. Three-way handshaking: Fig.2.2, p.37

   (a) The server must be prepared to accept connection requests (the *socket, bind, and listen* functions).

   (b) An *active open* is issued by the client using the *connect* call.

      . This function call sends a *SYN* (stands for synchronize) segment to inform the server the initial sequence number (for the data the client will subseqencely send).

      . Besides the initial seq number, the SYN segment just contains the IP header, TCP header, and possible TCP options

   (c) The server must acknowledge the client's SYN request. It also have to indicate its own initial sequence number. It does so by sending a single message: SYN K, ack $J + 1$.

   (d) The server must acknowledge the server's SYN message.

   (e) Notes: because an SYN occupies one byte of the sequence number space, the ack for both the client and server is one plus the indicated initial seq number.

b. Normal termination of a TCP connection: Fig.2.3, p.40

   (a) The application (normally the client) calls *close* function first. This is called an *active close*. The call triggers the TCP at its end to send a FIN segment, which meant that it finished sending data.

   (b) The other end that receives the FIN segment performs a so called *passive close*. The receiving of the FIN will trigger the TCP at that end to send an ack message. The receipt of the FIN also is passed to the application as an end-of-file condition.

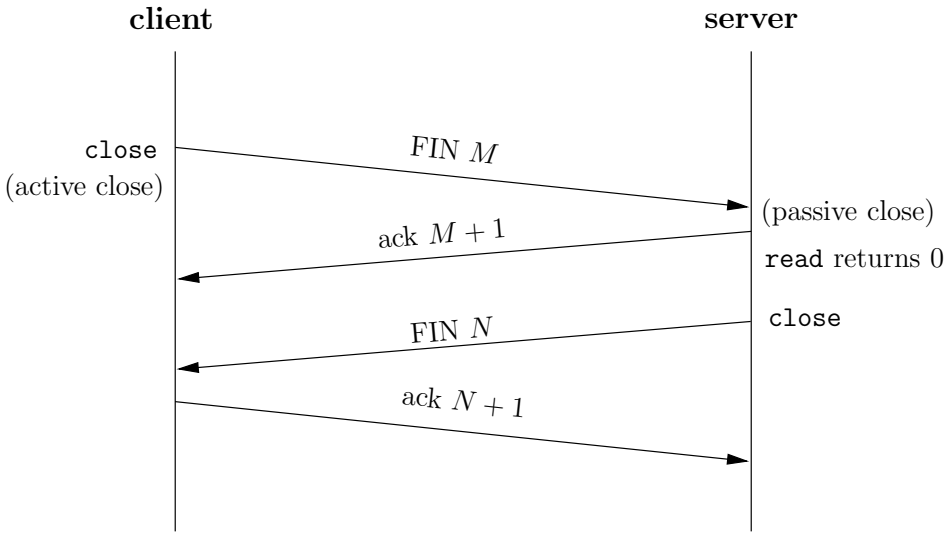


Figure 62: Packets exchanged when a TCP connection is closed (Fig.2.3,p.40)

(c) Sometime later the application that received the end-of-file condition will execute a *close* call, which triggers the TCP at that end to send an FIN segment to the other end.

(d) The TCP on the machine that receives this final FIN will send a final ack.

c. Notes about TCP termination

(a) Between Step ii and iii it is possible for data to flow from end doing the passive close to the other end. This is called *half-close* because the active close end only declares that it no longer sends data. (More in the *shutdown* function, which provides mechanism to control how to to terminate a connection).

(b) An FIN segment is sent when an end closes the corresponding socket. However, when a process terminates, either voluntarily or involuntarily, all open descriptors (including socket descriptors) are closed, which causes an FIN to be sent to every active TCP connection.

(c) Normally an TCP client will initiate the termination process. However, an TCP server can also initiate this process (such as HTTP server).

d. TCP state transition diagram: Fig.2.4, p.41

(a) There are 11 states, and they can all be displayed by using the *netstat* command.

(b) The end that performs *active close* will be in a so called *TIME_WAIT* state before it finally terminates.

e. Watching the packets: Fig.2.5,p.42, for a complete TCP connection session.

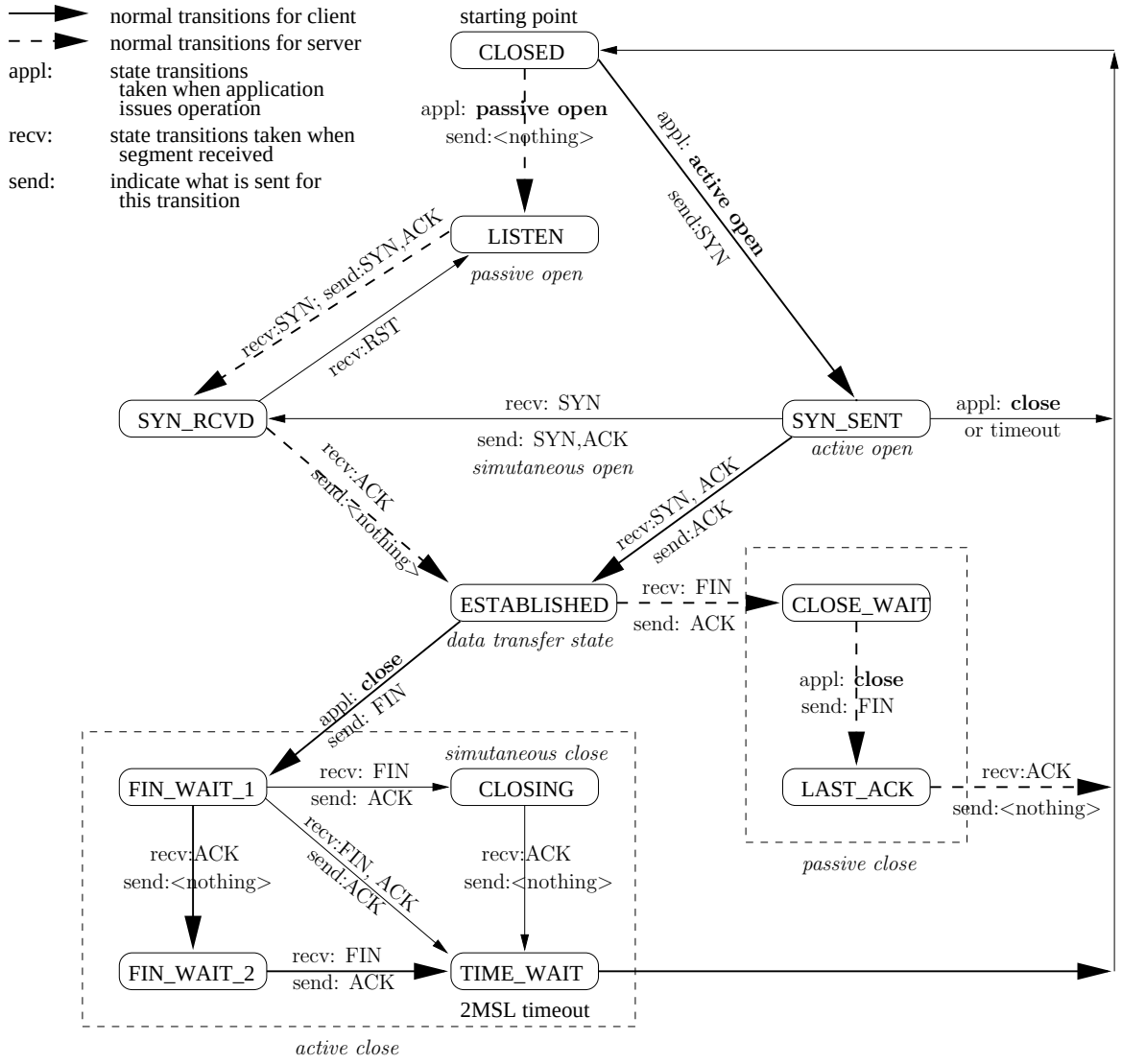


Figure 63: TCP state transition diagram (Fig.2.4,p.41)

(a) The client advertizes an 1460 MSS (typical when Ethernet is used), and the server responds by announcing an 1024 MSS. Notice that the MSS for the client and server can be different.

(b) The client sends its first request after the connection is established. This requests must be less than or equal to 1028 bytes.

(c) The server sends a reply to the client after receiving and processing the request. This reply must be less than 1460 bytes.

(d) *Piggybacking*: the acknowledgment to the client from the server is sent together with the reply to the client. This would normally happen if the server will take less than 20ms to process the request. Otherwise the ack and reply

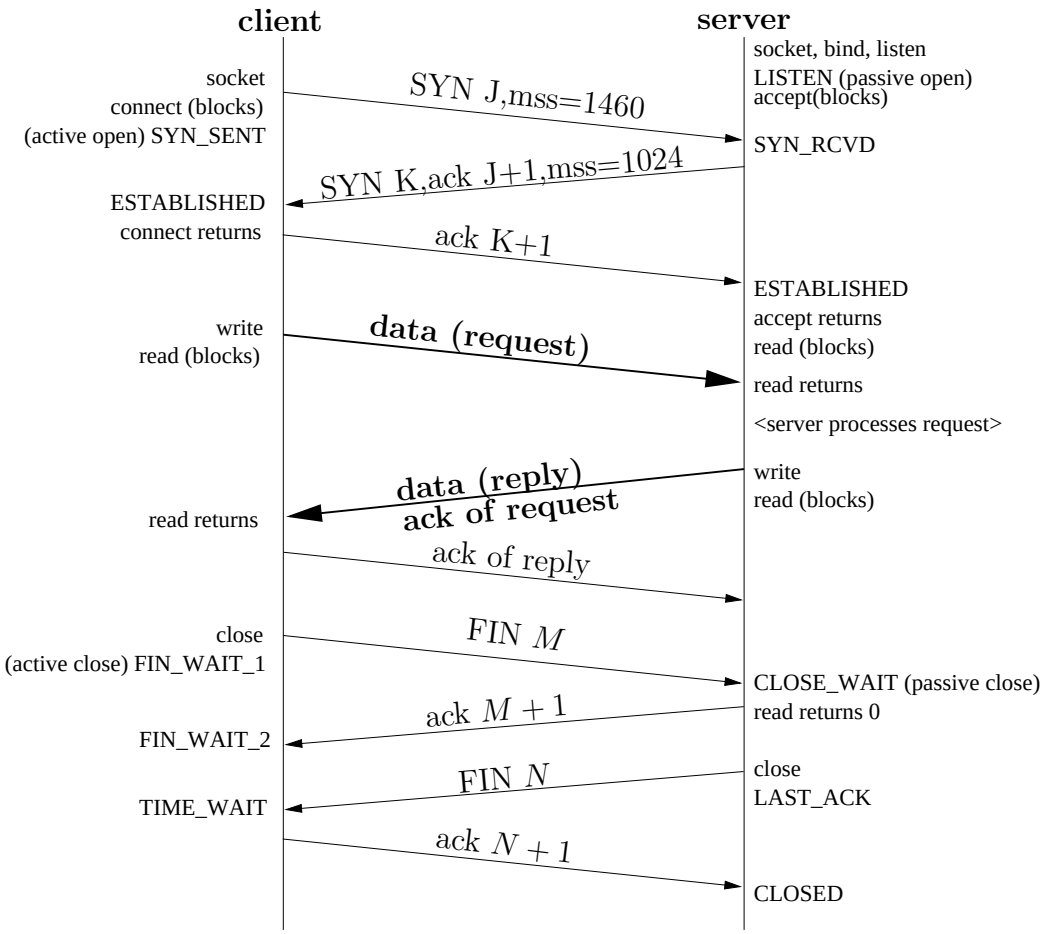


Figure 64: Packet exchanges for TCP connection (Fig.2.5,p.42)

to the client will be sent separately by the server.

(e) The termination part shows that the client enters the TIME_WAIT state.

(f) Overhead of TCP communications: To establish a TCP connection, seven TCP control segments have to be exchanged. If a client in TCP session only sends one request, then only two data segments are exchanged. The overhead is very subtantial. On the other hand, a UDP session in this case, only two UDP datagrams are exchanged.

f. The time_wait state

(a) MSL (maximum segment lifetime). Each implementation must choose an MSL and RFC 1122 recommends 2 min for MSL. BSD based implementation uses 30 sec.

(b) Therefore the 2MSL duration for the TIME_WAIT state will vary from 1 min to 4 min.

(c) Assumption: a packet with the maximum TTL (255 hops) value will not be able to survive on the internet for more 2MSL duration.

(d) *Lost duplicate* (or *wandering duplicate*): packets that are delayed for various reasons (routing anomaly, bugging software) may resurface on the internet late on.

(e) Reasons for *TIME_WAIT* state: (1) to implement TCP termination reliably; and (2)getting rid of the lost duplicates.

(f) Example 1: Assume that after the server receives the *FIN* message from the client, it sends the *ack* and *FIN*. The client then responded with its final *ACK*, which is lost and hence never reach the server. Therefore The server will resend its final FIN. Hence the client must maintain state info so that it can resend the final ACK. If it did not, it would have to send an RST as response (because the client has already closed its connection and the client kernel has no information about the previously terminated connection), which signals an error. For a TCP connection to terminate correctly, the end that initiates the termination must be handle loss of any of the final segments. Therefore the *TIME_WAIT* state is always on the end that initiates the termination.

(g) Example 2: a TCP connection at port 1500 of 206.62.226.33 and port 21 of 198.69.10.2. Assume that the connection is closed and sometime late a new connection is established using the same port numbers (1500 and 21) and same two IPs (called an incarnation of the previous connection). TCP must prevent packets from previous connections to reappear and be mis-interpreted in the new connection. TCP does this by not allowing connections that are still in TIME_WAIT state to be incarnated. The 2MSL will allow a packet in one direction to be lost and the duplicate reply lost in another direction.

(4) TCP options

a. The MSS (maximum segment size) option. Sent as part of the SYN message, MSS denotes the maximum amount of data that a TCP application is willing to accept in each TCP segment. Can be set or inspected by using the *TCP_MAXSEG* socket option.

b. Window scale option (a new option intended to support high-speed connections). The normal maximum window size is 65535 ($2^{16} - 1$, as the TCP header contains a 16-bit length field). This option can be an 1 to 14 bits integer and this value will allow the maximum window size to be as large as $65535 * (2^{14})$ (one gigabyte). Both ends must support this option for it to be used.

c. Timestamp option (aother new option intended to support high speed connections). It is used to prevent corrupted data caused by lost packets to reappear. Programmers do not have to worry about this option because it is used by the kernel.

(5) TCP header (from TCP/IP Illustrated Volume 1, Stevens)

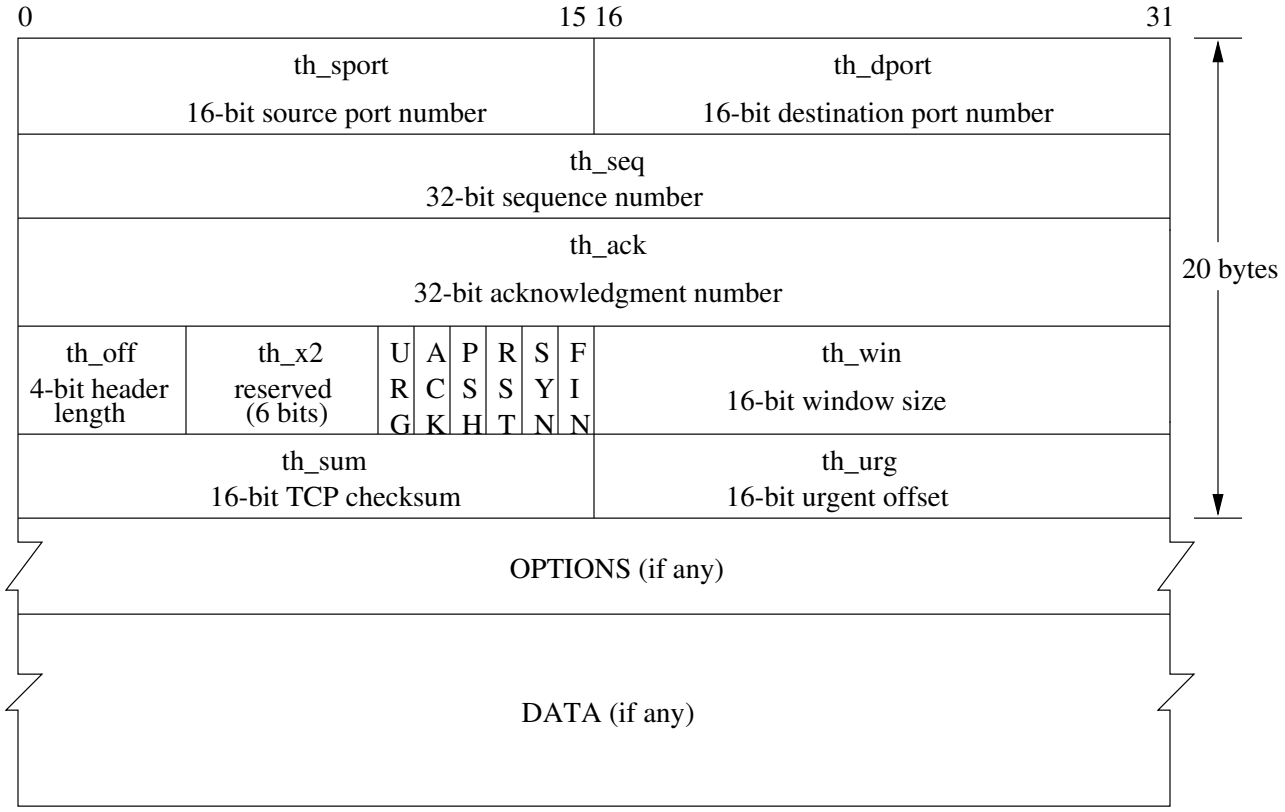


Figure 65: TCP Header and Optional Data

a. Can be up to 60 bytes (same as for IP header).

b. A 16-bit *source port number* field and a 16-bit *destination port number* field.

c. A 32-bit *sequence number* field.

d. A 32-bit *acknowlegment number* field.

e. A 4-bit *header length* field.

f. A 6-bit *reserved* field.

g. A 6-bit *flag* field.

   * URG bit: the urgent pointer (to be discussed) is valid;

- ACK bit: the acknowledgment number is valid;
- PSH bit: the receiver should pass this data to the application asap;
- RST bit: reset the connnection;
- SYN bit: synchronize sequence numbers to initiate a connection;
- FIN bit: the sender is finished sending data.

h. A 16-bit *window size* field. By advertising this value, TCP can perform *flow control*.

i. A 16-bit *TCP checksum size* field, which covers the entire TCP segment.

j. A 16-bit *TCP urgent pointer* field. The value of this pointer plus the seq number in the current segment will yield the seq number of the last byte of the urgent data. Notes: TCP itself will not do anything regarding to the urgent data. It's up to the applicaiton to decide how to locate and how to handle the urgent data.

k. Options that can be as long as 40 bytes.

7. Concurrent servers and port numbers

(1) Types of servers: *iterative* server and *concurrent* server

a. Iterative server: the server knows in advance how long it takes to finish the service requests and handles each request by itself.
Example. The Internet Daytime protocol.

b. Concurrent server: the server does not know in advance how long it takes to finish the service requests and handles each request by starting another process.

(2) Port numbers

a. From client to server: what services does the client want?

b. From server to client: how does the server send back message to the client?

c. A TCP or UDP header contains a 16-bit port number, which is assigned by the client machine.

d. TCP, UDP port numbers 1-255 are reserved. BSD4.3 reserves 1-1023. Fig. 2.10, p.51

e. Five-tuple that defines an association (of a client and a server):

< protocol, local host addr, local port#, foreign host addr, foreign port # >

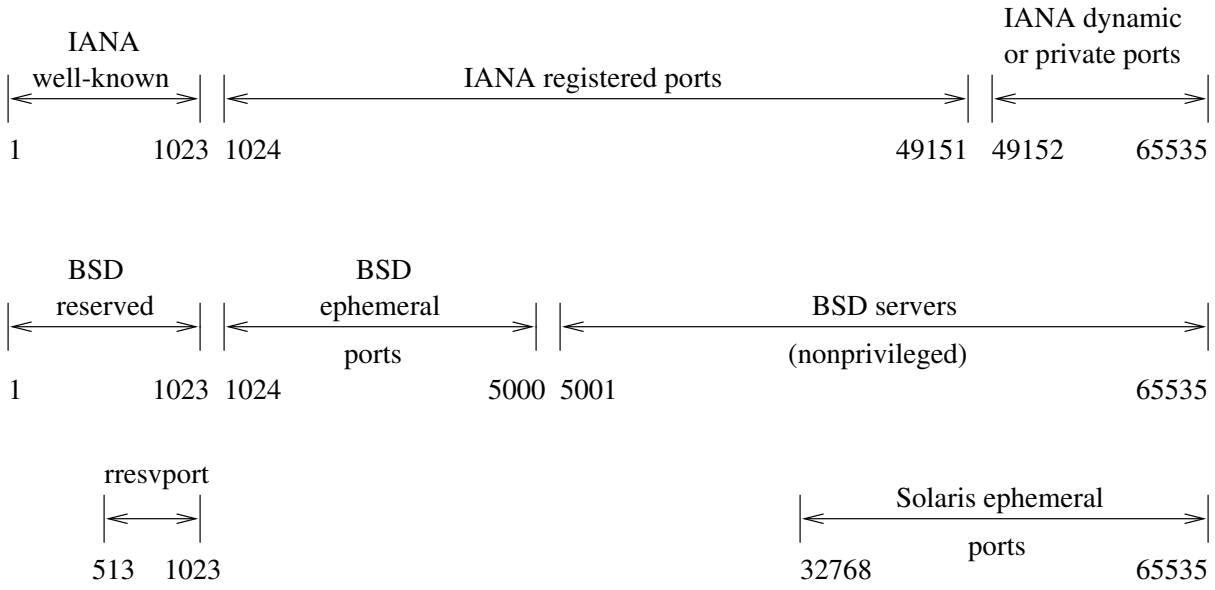


Figure 66: Allocation of port numbers (Fig.2.10,p.51)

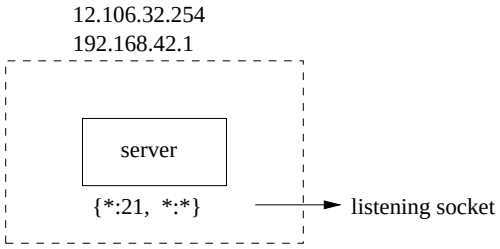


Figure 67: TCP server with a passive open on port 21 (Fig.2.11,p.53)

f. **Example.** Fig.2.11, Fig.2.12, p.53, Fig.2.13, p.54, and Fig.2.14, p.55.

(3) The roles of the client and server processes are asymmetric (they are coded differently).

a. The server is started first and typically does the following steps.

(a) Open a communication channel and inform the local host of its willingness to accept client requests on some well-known address.

(b) Wait for a client request to arrive at the well-known address.

(c) For an iterative server, process the request and send the reply (typically a request can be handled with a single response by such a server).

(d) For a concurrent server, a new process is spawned to handled the client request (*fork* and possibly *exec* under Unix is required). This new process just handles this client's request and closes its communication channel when finished).

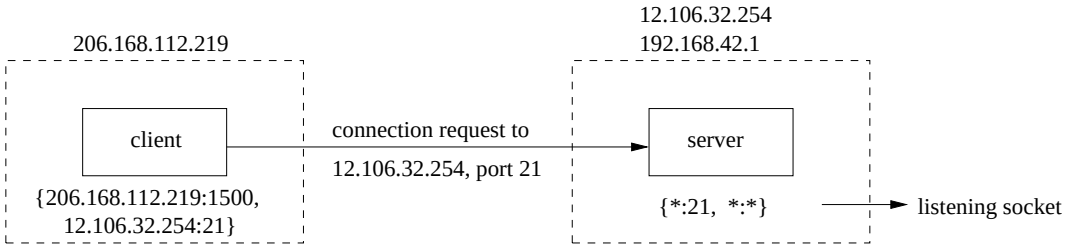


Figure 68: Connection request from client to server (Fig.2.12,p.53)

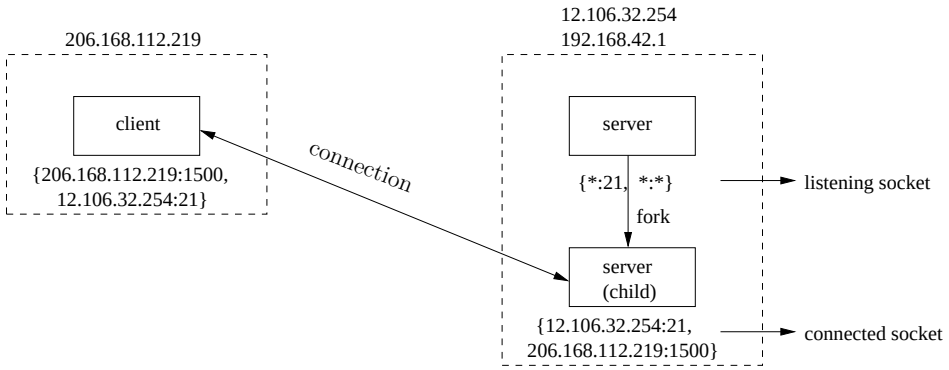


Figure 69: Concurrent server has child handle client (Fig.2.13,p.54)

Note: a concuccent server may use another *thread* to handle a new request instead.

(e) Go back to Step (b).

b. The client's actions are different:

(a) Open a communication channel and connect to a well-known address on a specific host (i.e. the server).

(b) Send service request messages to the server, and receive the responses. Continue doing this as long as necessary.

(c) Close the communication channel and terminate.

c. For the two different type of servers and TCP/UDP, we have the following combinations:

	iterative server	concurrent server
connection-oriented protocol	infrequent (DayTime)	typical
connectionless protocol	typical	infrequent (TFTP)

8. Buffer sizes and limitations

(1) Limitations

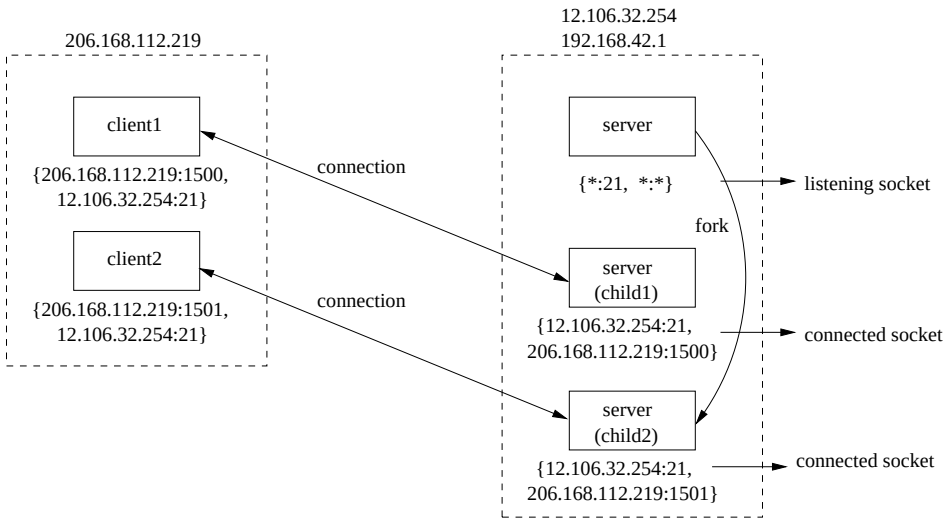


Figure 70: Second client connection with same server (Fig.2.14,p.55)

a. Maximum size of an IPv4 datagram: 65535 bytes, including IPv4 header.

b. The maximum size of an IPv6 datagram is 65575 bytes, including the 40-byte header.

c. MTU is maximum transmission unit that a network can support. For networks that use Ethernet as LAN, the MTU has to be smaller than or equal to 1500. The smallest MTU for IPv4 is 68 and the smallest MTU on IPv6 is 576.

d. Path MTU is the smallest MTU on the path between two hosts. The Ethnet MTU is often today's path MTU.

e. If the size of an IP datagram is larger than the path MTU, IPv4 and IPv6 will perform fragmentation. The framents will not be reassembled until they reach the destination.

f. The DF bit (don't fragment) bit allows the IPv4 to specify that an IPv4 datagram should not be fragmented anywhere (either by the generating host or any intermediate routers). If an IP datagram cannot be delivered due to its size larger than the path MTU, the encounting IP router will generate an ICMPv4 message "destination unreachable, fragmentation needed but DF bit set".

g. IPv6 has the DF bit set by default.

h. Both IPv4 and IPv6 define a *minimum reassembly buffer size*, which is the minimum datagram size that we are guarantteed any implementation must support. For IPv4 this value is 576 and for IPv6 this value is 1500 (to support Ethernet MTU). That means that we have no idea whether a given destination can accept a

577-byte datagram or not. Therefore many IPv4 applications that use UDP (such as DNS, RIP, TFTP, BOOTP, SNMP) prevent applications from generating IP datagrams larger than this size.

(2) TCP output: what would happen when an application writes data to a TCP socket: Fig.2.15, p.58

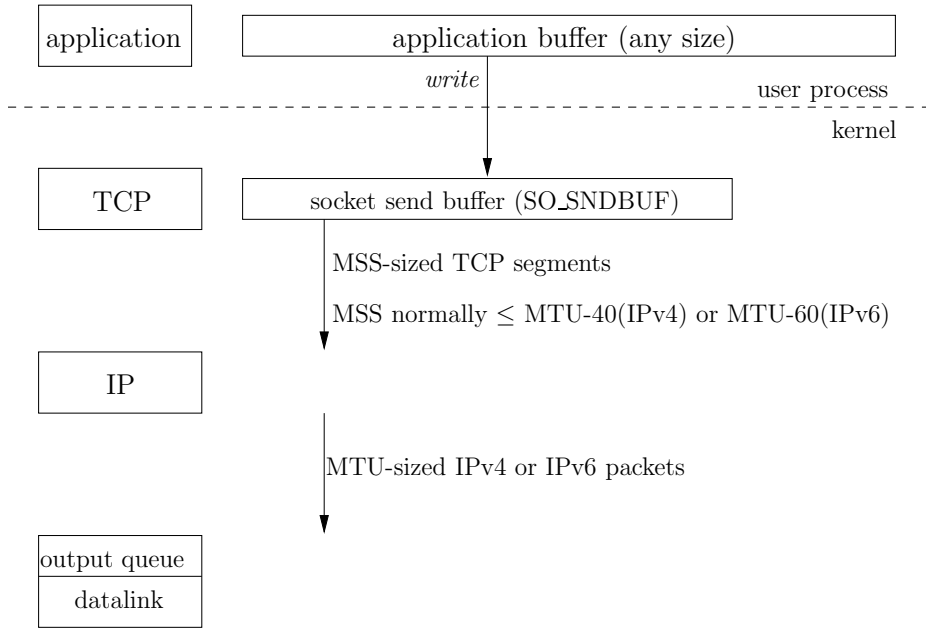


Figure 71: Steps and buffers involved when application writes to a TCP socket. (Fig.2.15,p.58)

a. A *send buffer* is associated with every TCP socket. The buffer size can be changed by using the *SO_SNDBUF* socket option.

b. When the application calls *write*, the kernel copies all the data from the application buffer into the socket send buffer. If the socket send buffer does not have sufficient free room, the application will be put into sleep (normal blocking semantics is assumed here). The write will not return until the last byte in the application buffer is copied to the socket send buffer.

c. TCP takes the data in the socket send buffer and sends it to the peer's TCP. TCP must keep the data until it is acknowledged by the peer.

d. TCP sends the data to IP in units of size MSS or smaller as announced by its peer (default is 536 if the peer does not send an MSS option). It will append its header to the segment.

e. IP will append its own header and performs routing and passes the IP datagram to the appropriate datalink.

f. IP will perform fragmentation if necessary. But the use of path MTU can reduce the need of fragmentation.

(3) UDP output: what would happen when an application writes data to a UDP socket: Fig.2.16, p.59

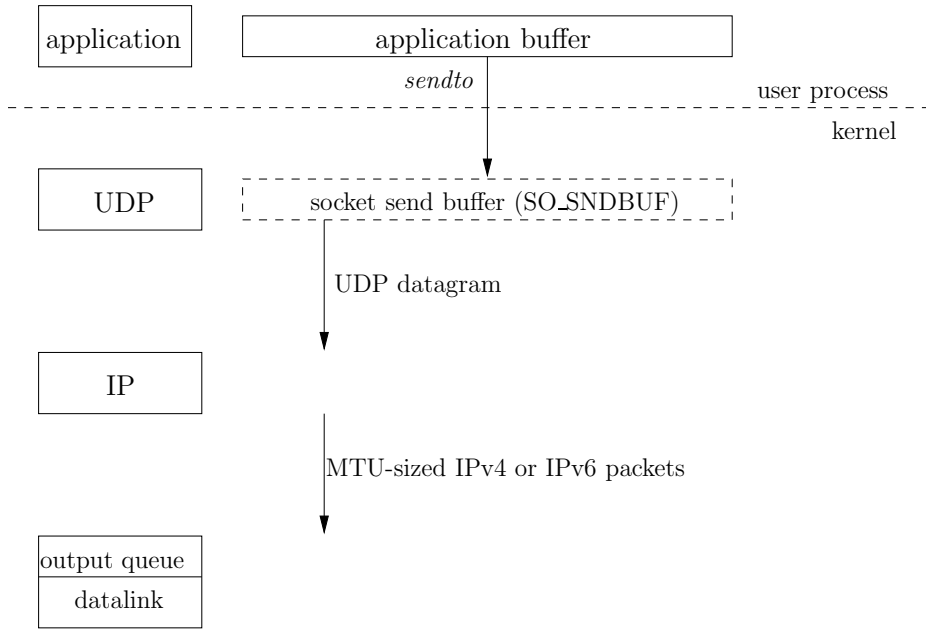


Figure 72: Steps and buffers involved when application writes to a UDP socket. (Fig.2.16,p.59)

a. For UDP, there does not exist a socket send buffer. The reason is that UDP is unreliable hence there is no need to buffer UDP datagram to keep state information for retransmission.

b. An UDP socket does have a send buffer size parameter. But that value is simply the maximum size of a UDP datagram that can be written to a socket. An EMSGSIZE error will occur if an application attempts to write a datagram larger than this limit.

c. UDP simply appends its 8-byte header and passes the datagram to IP.

d. UDP is more likely to perfrom fragmentation with larger (2000 bytes or above) datagrams than in TCP. The latter uses MSS sized segments.

e. A successful return of a writing to a UDP socket means that either the datagram or all fragments of the datagram have been added to the datalink queue. If no more room on the queue, an ENOBUFS error is returned to the application.

9. Standard (well known) Internet services

(1) Services whose port numbers are the same and known are called standard services.

(2) Example: Fig. 2.18, p.61: Some example services and their port numbers.

Name	TCP port	UDP port	RFC	Description
echo	7	7	862	Server returns whatever the client sends
discard	9	9	863	Server discards whatever the client sends
daytime	13	13	867	Server returns the time and date in a human-readable format.
chargen	19	19	864	Server sends a continual stream of characters, until the connection is terminated by the client. UDP server sends a datagram containing a random number of characters (between 0 and 512) each time the client sends a datagram.
time	37	37	868	Server returns the time as a 32-bit binary number. This number represents the number of seconds since midnight January 1, 1900, UTC.

Figure 2.18 Standard TCP/IP services provided by most implementations.

(3) The file /etc/services constains mapping of service names, port numbers, and service protocols used.

(4) Protocols used by common Internet applications: Fig.2.19, p.62.

Application	IP	ICMP	UDP	TCP	SCTP
ping		•	•		
traceroute		•			
OSPF (routing protocol)	•				
RIP (routing protocol)			•		
BGP (routing protocol)				•	
BOOTP (bootstrap protocol)			•		
DHCP (bootstrap protocol)			•		
NTP (time protocol)			•		
TFTP (trivial FTP)			•		
SNMP (network management)			•		
SMTP (electronic mail)				•	
Telnet (remote login)				•	
SSH (secure remote login)				•	
FTP				•	
HTTP (the Web)				•	
NNTP (network news)				•	
LRP (remote printing)				•	
DNS (domain name service)			•	•	
NFS (network file system)			•	•	
SUN RPC			•	•	
DCE RPC			•	•	
IUA (ISDN over IP)					•
M2UA,M3UA (SS7 telephony signaling)					•
H.248 (media gateway control)			•	•	•
H.323 (IP telephony)			•	•	•
SIP (IP telephony)			•	•	•

Figure 2.19 Protocol usage of various common Internet applications.

10. Similarities and differences between file I/O and network I/O

(1) Main functions for file I/O: *open, creat, close, read, write, lseek*, and *dup*. All of them carry a *file descriptor*.

(2) Ideally, the networking operation has the same semantics as file I/O. Practically, network I/O is more complex than file I/O:

a. The typical client-server relationship is not symmetrical. Must decide which role to play in a networking operation.

b. Network operations can be connection-oriented or connectionless. The former case is more like file I/O than the latter.

c. Names are more important in networking than in file I/O. File descriptors can be passed from parent to child processes and they are all the child processes need to access files. In networking, child processes may have to know the names of other hosts and processes in order to be authenticated.

d. An association (of a client and a serve) is a five-tuple (cf. port number concept in UDP introduction):

$$< \text{protocol, local-addr, local-process, foreign-addr, foreign-process} >$$

Each of these parameters can differ from one protocol to another.

e. Record boundaries are important to some protocols, while TCP protocol is stream-oriented.

f. The network interface should support multiple comm. protocols. For instance, it cannot just use a 32-bit integer for network addresses (ok for internet, not sufficient for XNS addresses).

11. Sockets and socket structures (Chapter 3, 2nd textbook)

(1) Sockets

a. A socket is an end point of a communication session.

(a) In programming a socket is represented by a *socket descriptor*. Similar to file descriptors, a socket descriptor value is a non-negative integer.

(b) Before a communication session can take place, a process must first obtain a socket descriptor that has been assigned a non-negative integer value.

(c) Socket descriptors are system resources. Each kernel can only maintain/support a predefined maximum number of socket descriptor values.

b. As a socket represents an end point of a communication session, and each communication session has its own properties. These properties are specified in a socket structure.

(2) Socket structures.

a. Many of the BSD networking functions require a pointer to the following socket address structure as an argument. For TCP socket, the following structure is defined in <netinet/in.h>:

```
struct in_addr {
    in_addr_t       s_addr;         /* 32-bit IPv4 address */
                                    /* network byte ordered */
};
struct sockaddr_in {
    uint8_t         sin_len;        /* length of structure (16) */
    sa_family_t     sin_family;     /* AF_INET */
    in_port_t       sin_port;       /* 16-bit port number */
                                    /* network byte ordered */
    struct in_addr  sin_addr;       /* 32-bit IPv4 address */
                                    /* network byte ordered */
    char            sin_zero[8];    /* unused */
};
```

Figure 3.1 The Internet (IPv4) socket address structure: `sockaddr_in`

a. Notes: The length field was added with 4.3BSD-Reno to simplify handling of variable-length socket address structures (More in next topic).

b. POSIX.1g datatypes (Fig.3.2, p.69):

Datatype	Description	Header file
int8_t	signed 8-bit integer	<sys/types.h>
uint8_t	unsigned 8-bit integer	<sys/types.h>
int16_t	signed 16-bit integer	<sys/types.h>
uint16_t	unsigned 16-bit integer	<sys/types.h>
int32_t	signed 32-bit integer	<sys/types.h>
uint32_t	unsigned 32-bit integer	<sys/types.h>
sa_family_t	address family of socket addr struct	<sys/socket.h>
socklen_t	length of socket addr struct (normally uint32_t)	<sys/socket.h>
in_addr_t	IPv4 address (normally uint32_t)	<netinet/in.h>
in_port_t	TCP or UDP port (normally uint62_t)	<netinet/in.h>

Figure 3.2 Datatypes required by the POSIX specification.

c. Obsolete datatypes:

C Data type	BSD4.3	System V
unsigned char	u_char	unchar
unsigned short	u_short	ushort
unsigned int	u_int	uint
unsigned long	u_long	ulong

(3) Generic socket address structure

a. Motivations

* Socket API supports many different protocol families.
* Many socket related functions have to use some type of socket address structure. Socket address structure is normally passed by reference.
* Problem: do we have to define one function for each protocol family?
* Solution: use a generic socket address structure and type casting.

b. Structure definition: defined in <sys/socket.h>:

```
struct sockaddr {
    uint8_t       sa_len;
    sa_family_t   sa_family;      /* address family: AF_xxx value */
    char          sa_data[14];    /* protocol-specific address*/
};
```

Figure 3.3 The generic socket address structure: `sockaddr`

(4) IPv6 socket address structure (p.71, Fig.3.4)

a. Definition:

```
struct in6_addr {
    in6_addr_t        s6_addr;          /* 128-bit IPv6 address */
                                        /* network byte ordered */
};
#define SIN6_LEN
struct sockaddr_in6 {
    uint8_t           sin6_len;         /* length of this struct (28) */
    sa_family_t       sin6_family;      /* AF_INET6 */
    in_port_t         sin6_port;        /* transport layer port number */
                                        /* network byte ordered */
    uint32_t          sin6_flowinfo;    /* flow information, undefined */
    struct in6_addr   sin6_addr;        /* IPv6 address */
                                        /* network byte ordered */
    uint32_t          sin6_scope_id     /* set of interfaces for a scope */
};
```

Figure 3.4 IPv6 socket address structure: `sockaddr_in6`

b. Notes:

* The SIN6_LEN constant must be defined if the system supports the length member in the structure.
* The protocol family for IPv6 is AF_INET6.
* 64-bit alignment is supported for sockaddr_in6 if sockaddr_in is 64-bit aligned to optimize data access.
* The sin_flowinfo member is new and is still unused. It consists of three fields: (1) The low-order 24 bits are for flow label; (2) The next 4 bits are for priority; (3) The last 4 bits are reserved.

(5) New generic socket address structure

a. The new generic socket address structure is named `sockaddr_storage`. It was proposed as part of IPv6 to remedy the problem of the classical generic socket structure, which does not have enough storage space for some socket types.

b. It should be defined in <netinet/in.h>:

```
struct sockaddr_storage {
    uint8_t       sa_len;     /* length of this struct (implementation dependent) */
    sa_family_t   sa_family;  /* address family: AF_xxx value */
    /* implementation-dependent elements to provide:
     * a) alignment sufficient to fulfill the alignment requirements of
     *    all socket address types that the system supports.
     * b) enough storage to hold any type of socket address that the
          sytem supports.
     */
};
```

Figure 3.5 The storage socket address structure: `sockaddr_storage`

(6) Comparison of socket address structures: Fig.3.6, p.73

a. The sockaddr_un is for UNIX domain socket and is discussed in Chapter 15.

b. The sockaddr_dl is for UNIX datalink socket and is discussed in Chapter 18 (Routing sockets).

12. Utility functions

(1) Value-Result Arguments

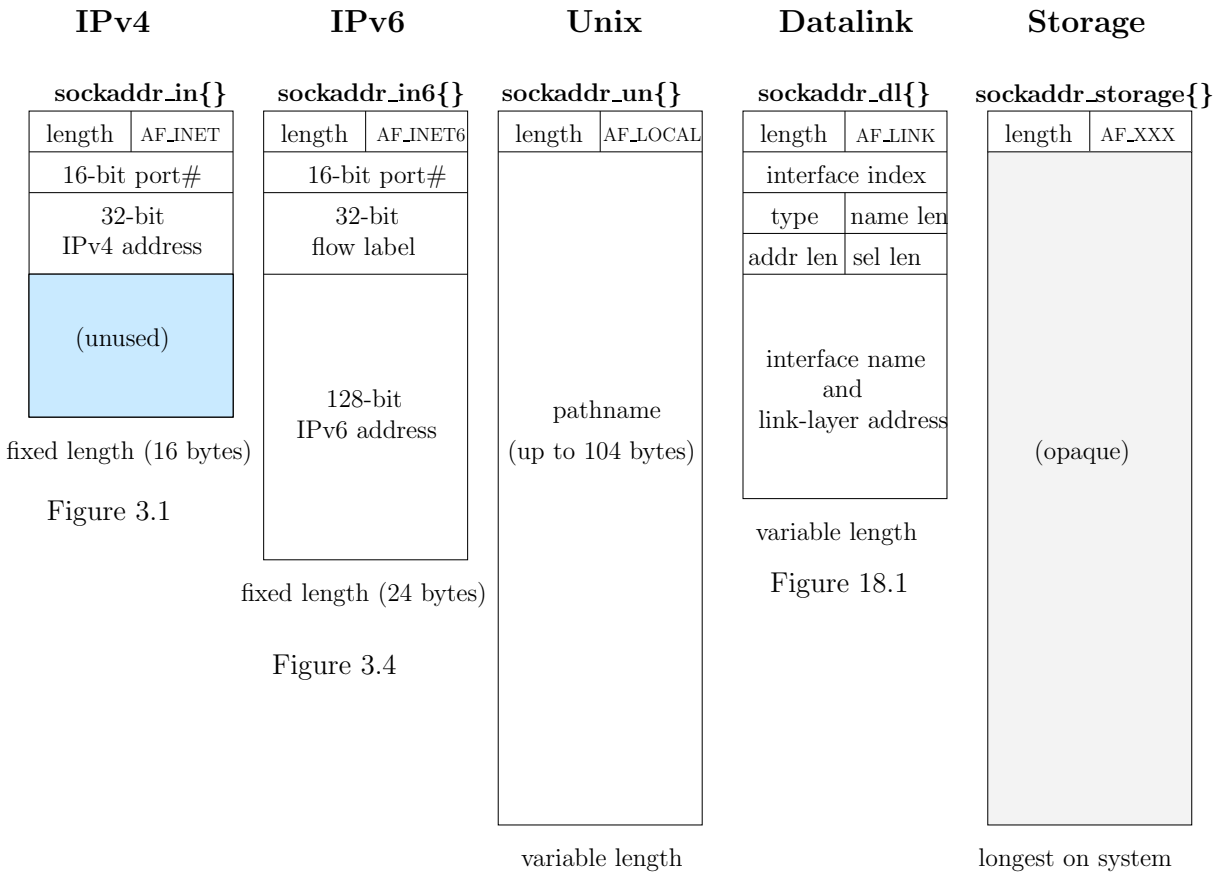


Figure 73: Comparison of various socket address structures (Fig.3.6,p.73)

a. Passing by value: Example 1: the last parameter of functions *connect, bind*. Fig.3.6, p.64.

b. Passing by reference: Example: the second parameter of functions *connect*;

c. Passing by value-result;

   (a) The function *getpeername*: p.64 (to be discussed in detail in next lecture):

```
struct sockaddr_un cli;        /* Unix domain */
socklen_t len;
len = sizeof (cli);            /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */
```

   (b) The last two parameters are *value-result* parameters.

   (c) Processes pass value to kernel: Fig.3.7, p.75

   (d) Kernel passes socket structure and length of that structure to user process:

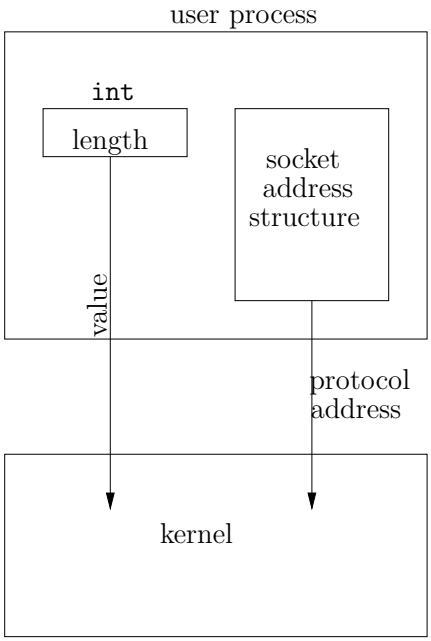


Figure 74: Socket address structure passed from process to kernel (Fig.3.7,p.75)

Fig.3.8, p.76

d. Other functions that use value-result arguments: the length argument of function *recvfrom*, function *accept*, function *getsockname*; (p.66, top) middle three arguments of function *select*; the length argument of function *getsockopt*; the msg_namelen and msg_controllen member of *msghdr* structure when used with *recvmsg* function;

(2) Byte ordering routines – four functions handling the potential byte order differences between different computer architecture and network protocols.

a. Basic concepts: p.77, Fig.3.9.

b. Little-endian byte order, big-endian order, and host byte order

(a) Little-endian order: for a 2-byte word, the low-order byte A is at lower memory address, the high-order A+1 is at higher memory address.

(b) Big-endian order: for a 2-byte word, the low-order byte A is at higher memory address, the high-order A+1 is at lower memory address.

(c) Host byte order: used on a specific machine.

c. A program testing host byte order: Fig.3.10, p.78

```
1  #include   "unp.h"
```

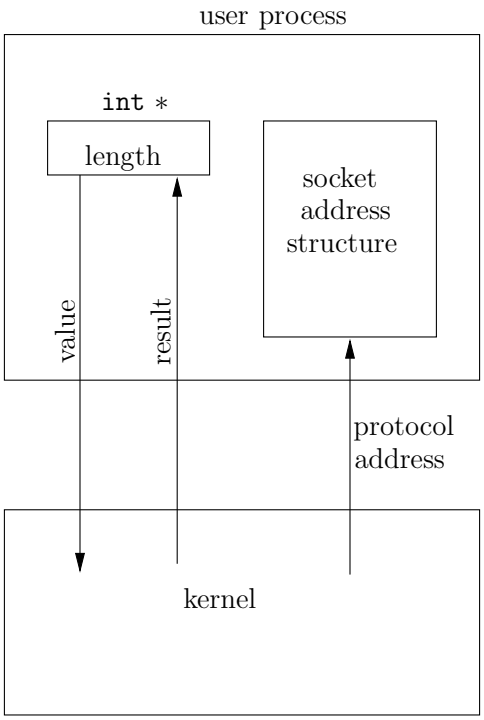


Figure 75: Socket address structure passed from kerne to process (Fig.3.8,p.76)

```
2  int
3  main(int argc, char **argv)
4  {
5     union {
6       short  s;
7       char   c[sizeof(short)];
8     } un;

9     un.s = 0x0102;
10    printf("%s: ", CPU_VENDOR_OS);
11    if (sizeof(short) == 2) {
12       if (un.c[0] == 1 && un.c[1] == 2)
13          printf("big-endian\n");
14       else if (un.c[0] == 2 && un.c[1] == 1)
15          printf("little-endian\n");
16       else
17          printf("unknown\n");
18    } else
19       printf("sizeof(short) = %d\n", sizeof(short));
```

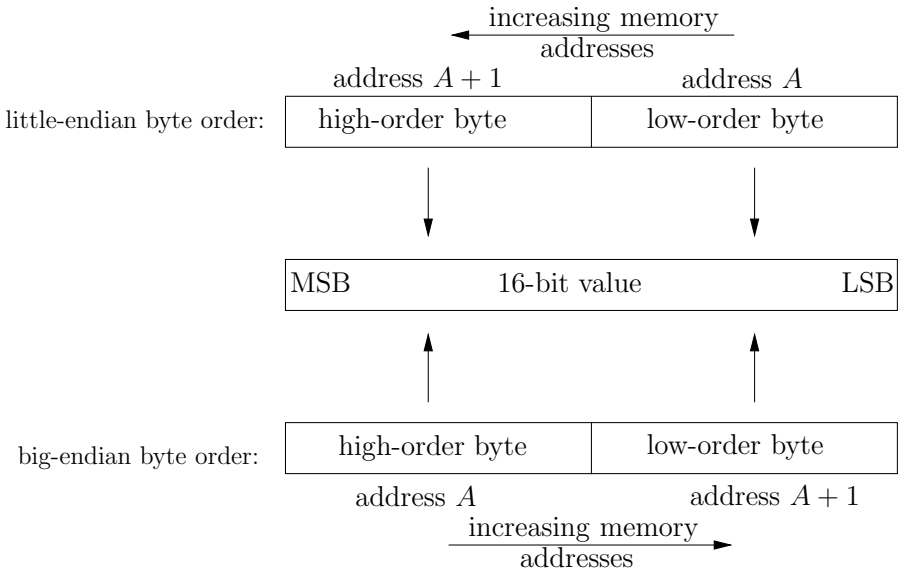


Figure 76: Little-endian byte order and bit-endian byte order for a 16-bit integer. (Fig.3.9,p.77)

```
20   exit(0);
21 }
```

Figure 3.10 Program to determine host byte order.

d. Byte ordering functions

```
#include <sys/types.h>
#include <netinet/in.h>

uint16_t htons(uint16_t host16bitvalue);
    /* convert host-to-network, 16-bit integer */

uint32_t htonl(uint32_t host32bitvalue);
    /* convert host-to-network, 32-bit integer */

uint16_t ntohs(uint16_t net16bitvalue);
    /* convert network-to-host, 16-bit integer */

uint32_t ntohl(uint32_t net32bitvalue);
    /* convert network-to-host, 32-bit integer */
```

All designed for Internet protocols. Operate on unsigned integers (but also work in signed integers).

(3) Byte manipulation operations – functions operating on *user-defined* byte strings. These are not C character strings, which always end with a null byte.

- void bzero(void **dest*, size_t *nbytes*);
- void bcopy(const void **src*, void **dest*, size_t *nbytes*);
- void bcmp(const void **prt1*, void **prt2*, size_t *nbytes*);
  This function returns zero if the two byte strings are identical, nonzero otherwise. Slightly different from the value returned by *strcmp* function.

(4) Address conversion functions *inet_aton, inet_addr*, and *inet_ntoa*

a. These functions convert between Internet addresses in network integer format and in dot-decimal format.

b. Function definitions

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

in_addr_t inet_addr(const char *strptr);
char *inet_ntoa(struct in_addr inaddr);
char *inet_aton(const char *strptr);
```

(a) The function `inet_addr` converts a character string in dot-decimal form to a 32-bit Internet address (as return value).

(b) The function `inet_ntoa` converts a 32-bit Internet address to a character string in dot-decimal form (as return value).

(c) The function `inet_aton` is almost the same as function `inet_addr`, except that the converted value is stored in a place pointed by the second parameter (supported by Linux. Not supported by Tru-64 UNIX or Sun Solaris).

(5) The *inet_pton* and *inet_ntop* functions.

a. Introduced in IPv6, but work for both IPv4 and IPv6.

b. Function definition: p.83.

```
#include <arpa/inet.h>

int inet_pton(int family, const char *strptr, void *addrptr);
    /* returns 1 if OK, 0 if output not a valid presentation format, -1 on error */
```

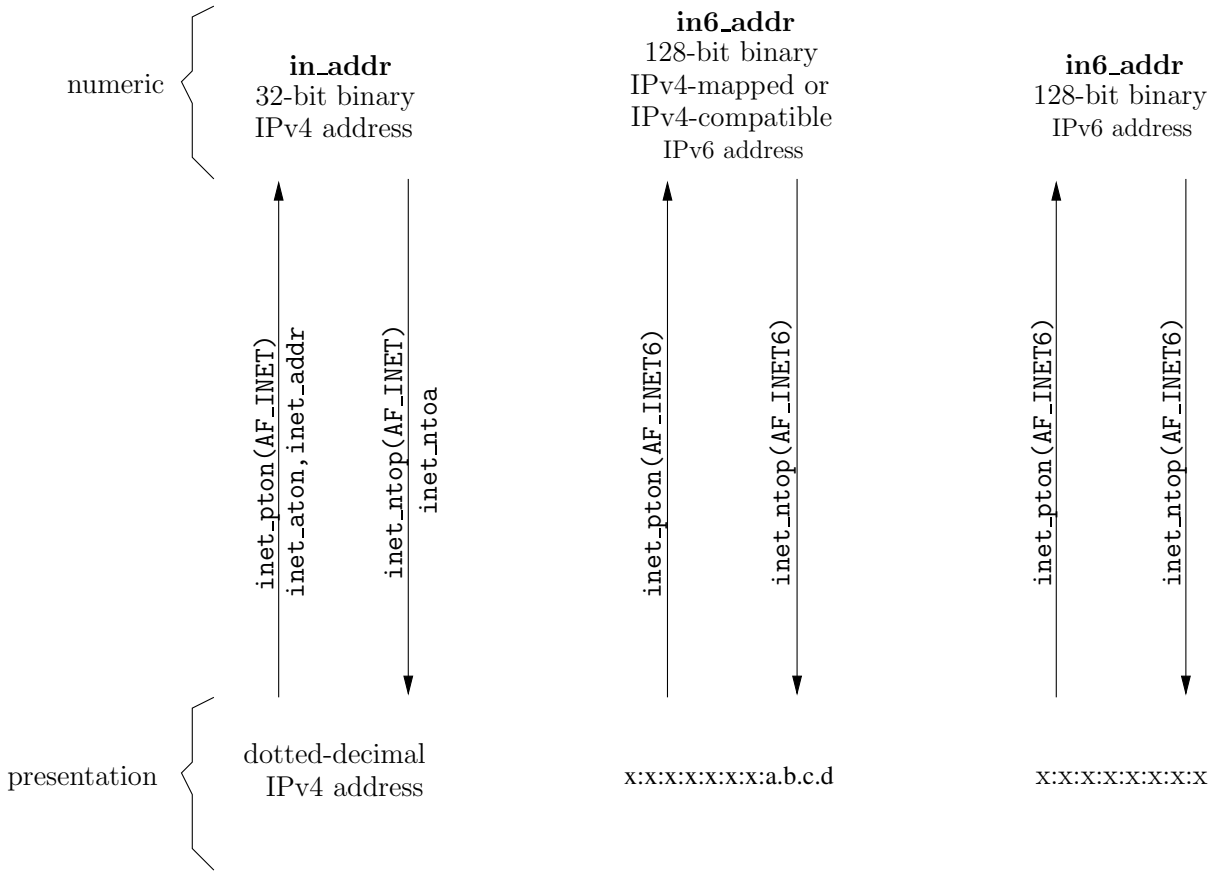


Figure 77: Summary of address conversion functions (Fig.3.11,p.84)

int `inet_ntop`(int *family*, `const void` ∗*addrptr*, `char` ∗*strptr*, `size_t` *len*);
/∗ returns pointer to result if OK, `NULL` on error ∗/

c. Summary of address conversion functions: Fig.3.11,p.84

(6) `sock_ntop` and related functions

a. Problem with the function `inet_ntop`:

(a) The second parameter of this function is a pointer that usually points to a structure field inside a socket sturcture.

(b) Due to the difference in socket structures, the caller has to know the type of the sockets to reference correctly that field in the socket structure.

(c) This makes the use of this function protocol dependent. This defeats the purpose of the function. For IPv4, we have to:

```
struct sockaddr_in addr;
inet_ntop(AF_INET, &addr.sin_addr, str, sizeof(str));
```

For IPv6, we have to code like:

```
struct sockaddr_in6 addr6;
inet_ntop(AF_INET6, &addr6.sin6_addr, str, sizeof(str));
```

b. The function `sock_ntop` is a function defined by the textbook that intends to solve the above problem.

(a) The function takes two parameters. The first parameter is a a pointer to a socket structure, and the second parameter is the length of the structure:

```
#include "unp.h"
char *sock_ntop(const struct sockaddr *sockaddr,socklen_t addrlen);
```

(b) The function will resolve the different socket types inside its implementations (by calling `inet_ntop` with different parameters), as shown in Fig.3.14, p.87.

c. There are several other similar user-defined functions on page 87.

(7) Utility functions *readn, writen, readline* (p.88-92)

a. Stream sockets exhibit a behavior with the *read* and *write* functions that differ from the normal file I/O. A *read* or *write* on a socket might input or output fewer bytes than requested, and this is not an error condition. This is due to buffer limits that might be reached for the socket in the kernel and all that is required is for the caller to invoke a read or write again.

b. Datagram sockets do not have this problem, because reading or writing is on the unit of a packet.

c. The three functions that can read/write a socket without having the un-needed read/write errors are shown in Fig.3.14 and Fig.3.15,p.78, Fig.3.16 on p.79

```
1  #include   "unp.h"

2  ssize_t                 /* Read "n" bytes from a descriptor. */
3  readn(int fd, void *vptr, size_t n)
4  {
5      size_t   nleft;
6      ssize_t   nread;
7      char   *ptr;

8      ptr = vptr;
9      nleft = n;
```

```
10    while (nleft > 0) {
11       if ( (nread = read(fd, ptr, nleft)) < 0) {
12          if (errno == EINTR)
13             nread = 0;      /* and call read() again */
14          else
15             return(-1);
16       } else if (nread == 0)
17          break;            /* EOF */

18       nleft -= nread;
19       ptr   += nread;
20    }
21    return(n - nleft);      /* return >= 0 */
22 }
```

Figure 3.15 `readn` function: read n bytes from a descriptor.

```
1  #include   "unp.h"

2  ssize_t                  /* Write "n" bytes to a descriptor. */
3  writen(int fd, const void *vptr, size_t n)
4  {
5     size_t      nleft;
6     ssize_t      nwritten;
7     const char   *ptr;

8     ptr = vptr;
9     nleft = n;
10    while (nleft > 0) {
11       if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
12          if (errno == EINTR)
13             nwritten = 0;      /* and call write() again */
14          else
15             return(-1);         /* error */
16       }

17       nleft -= nwritten;
18       ptr   += nwritten;
19    }
20    return(n);
21 }
```

Figure 3.16 `writen` function: write n bytes to a descriptor.

```
1  #include   "unp.h"

2  /* PAINFULLY SLOW VERSION -- example only */
3  ssize_t
4  readline(int fd, void *vptr, size_t maxlen)
5  {
6    ssize_t   n, rc;
7    char   c, *ptr;

8    ptr = vptr;
9    for (n = 1; n < maxlen; n++) {
10     again:
11       if ( (rc = read(fd, &c, 1)) == 1) {
12          *ptr++ = c;
13          if (c == '\n')
14             break;           /* newline is stored, like fgets() */
15       } else if (rc == 0) {
16          *ptr = 0;
17          return(n - 1);      /* EOF, n-1 bytes were read */
18       } else {
19          if (errno == EINTR)
20             goto again;
21          return(-1);         /* error, errno set by read() */
22       }
23   }

24   *ptr = 0;                 /* null terminate like fgets() */
25   return(n);
26 }
```

Figure 3.17 `readline` function: read a text line from a descriptor, one byte at a time.

d. Improved version of readline function: Fig.3.18,p.91-92

   (a) The first version of readline is not efficient because each call of read only reading one byte.

   (b) The improved version uses the standard read to read multiple bytes per read. The improved version is almost the same as the first version, except it calls the new function `my_read`, instead of the function `read`.

(c) The new function `readlinebuf` exposes the internal buffer state so that callers can check and see if more data was received beyond a single line.

```
1  #include   "unp.h"

2  static ssize_t
3  my_read(int fd, char *ptr)
4  {
5     static int   read_cnt = 0;
6     static char   *read_ptr;
7     static char   read_buf[MAXLINE];

8     if (read_cnt <= 0) {
9       again:
10       if ( (read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
11          if (errno == EINTR)
12             goto again;
13          return(-1);
14       } else if (read_cnt == 0)
15          return(0);
16       read_ptr = read_buf;
17    }

18    read_cnt--;
19    *ptr = *read_ptr++;
20    return(1);
21 }

22 ssize_t
23 readline(int fd, void *vptr, size_t maxlen)
24 {
25    int      n, rc;
26    char   c, *ptr;

27    ptr = vptr;
28    for (n = 1; n < maxlen; n++) {
29       if ( (rc = my_read(fd, &c)) == 1) {
30          *ptr++ = c;
31          if (c == '\n')
32             break;           /* newline is stored, like fgets() */
33       } else if (rc == 0) {
```

```
34          *ptr = 0;
35          return(n - 1);      /* EOF, n-1 bytes were read */
36       } else
37          return(-1);         /* error, errno set by read() */
38    }

39   *ptr = 0;                  /* null terminate like fgets() */
40   return(n);
41 }

42 ssize_t
43 readlinebuf(void **vptrptr)
44 {
45    if (read_cnt)
46       *vptrptr = read_ptr;
47    return (read_cnt);
48 }
```

Figure 3.18 Better version of `readline` function.