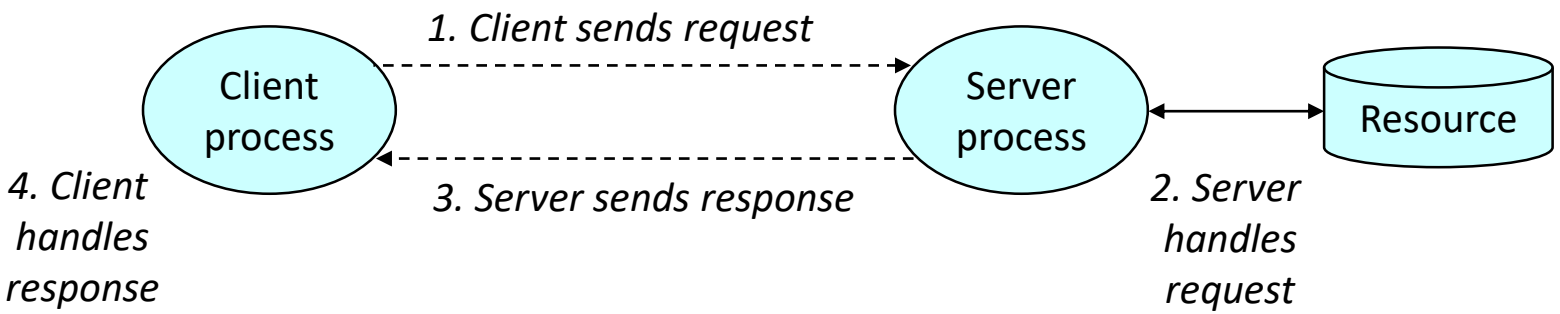
Programmer’s view of the Internet

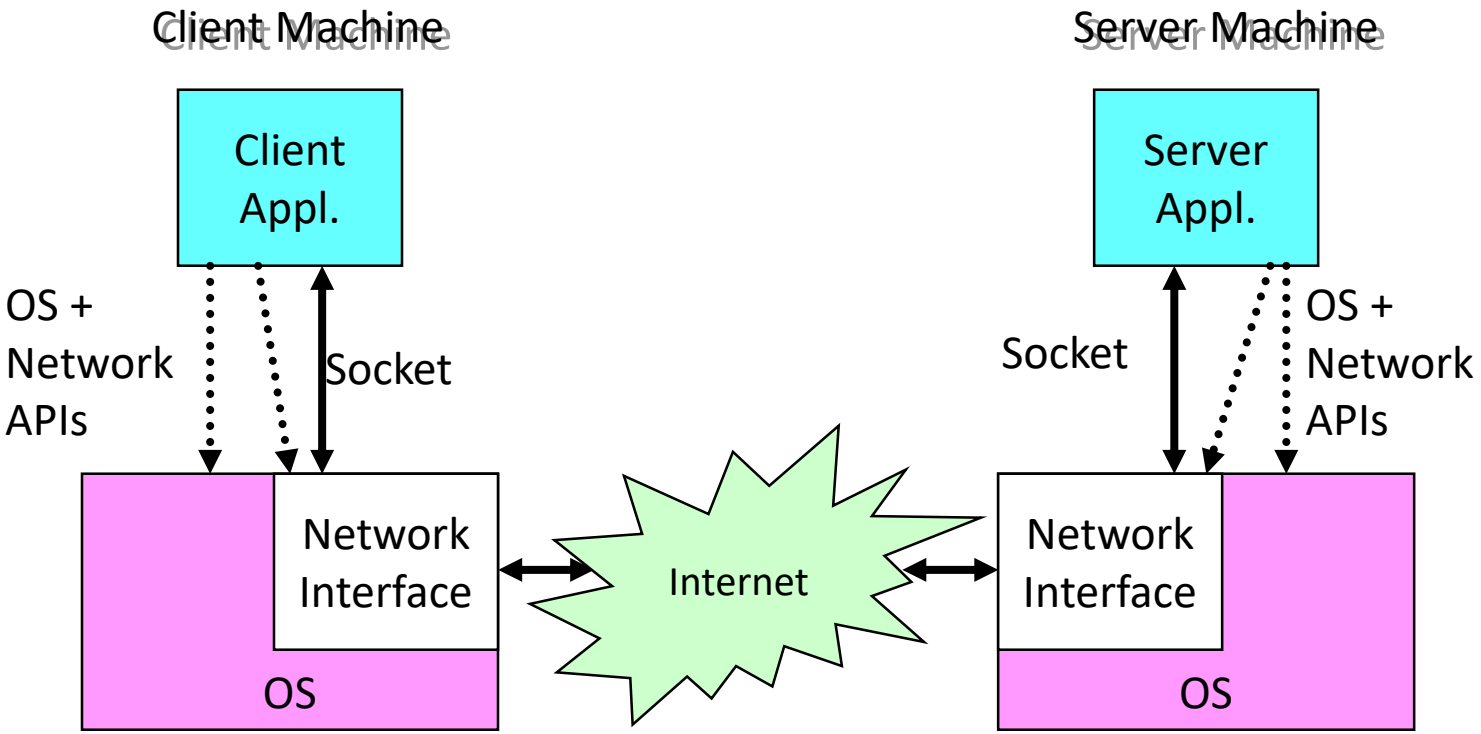
A Client-Server Exchange

- A *server* process and one or more *client* processes
- Server manages some *resource*.
- Server provides *service* by manipulating resource for clients.



Note: clients and servers are processes running on hosts (can be the same or different hosts).

Network Applications



Access to Network via Program Interface

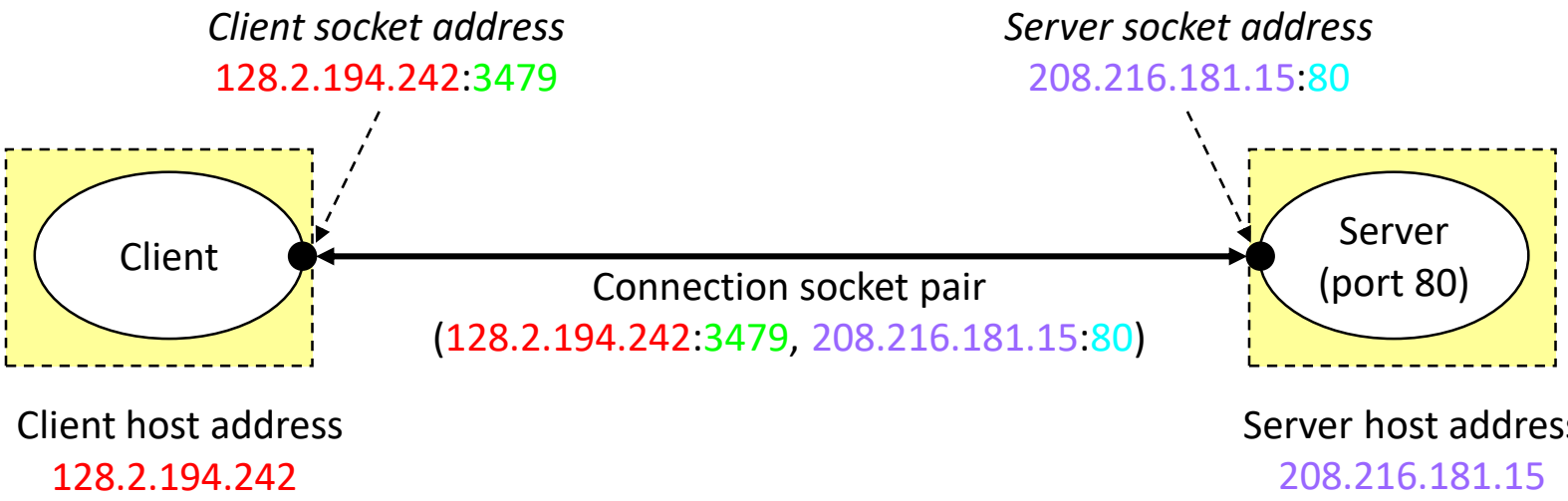
- Sockets make network I/O look like files
- Call system functions to control and communicate
- Network code handles issues of routing, segmentation.

Internet Connections (TCP/IP)

Two common paradigms for clients and servers communication

- Datagram Socket(UDP protocol SOCK_DGRAM)
- Stream Socket (TCP protocol, SOCK_STREAM)

Connections are point-to-point, full-duplex (2-way communication), and reliable.



Clients

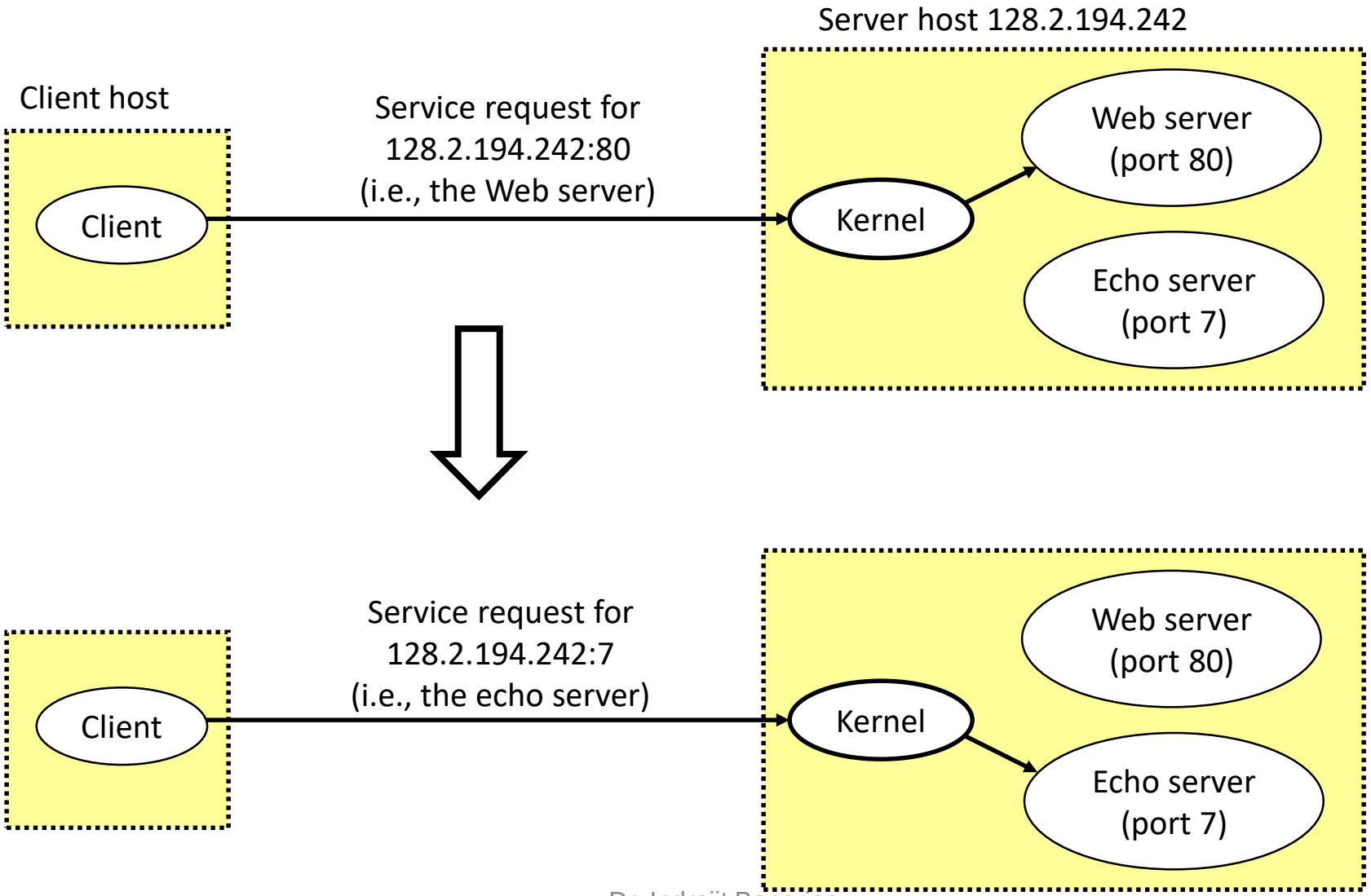
Examples of client programs

- Web browsers, `ftp`, `telnet`, `ssh`

How does a client find the server?

- The IP address in the server socket address identifies the host *(more precisely, an adaptor on the host)*
- The (well-known) port in the server socket address identifies the service, and thus implicitly identifies the server process that performs that service.
- Examples of well known ports
  - Port 7: Echo server
  - Port 23: Telnet server
  - Port 25: Mail server
  - Port 80: Web server

Using Ports to Identify Services



Servers

- Servers are long-running processes (daemons).
  - Created at boot-time (typically) by the init process (process 1)
  - Run continuously until the machine is turned off.
- Each server waits for requests to arrive on a well-known port associated with a particular service.
  - Port 7: echo server
  - Port 23: telnet server
  - Port 25: mail server
  - Port 80: HTTP server
- A machine that runs a server process is also often referred to as a “server.”

See `/etc/services` for a comprehensive list of the services available on a Linux machine.

Sockets Introduction

Socket Address Structures

- An IPv4 socket address structure, commonly called an "Internet socket address structure,"
- It is named sockaddr_in and is defined by including the <netinet/in.h> header

```
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 TCP or UDP port number */
                                 /* network byte ordered */
  struct in_addr   sin_addr;     /* 32-bit IPv4 address */
                                 /* network byte ordered */
  char             sin_zero[8];  /* unused */
};
```

Datatypes required by the POSIX specification

Datatype	Description	Header
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 address structure	<sys/socket.h>
socklen_t	Length of socket address structure, 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 uint16_t	<netinet/in.h>

Generic Socket Address Structure

- How to declare the type of pointer that may accept any type of pointer?
- With ANSI C, the solution is simple: void * is the generic pointer type
- The socket functions when take an argument it is defining a variable of *generic* socket address structure define in the <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 */
};
```

IPv6 Socket Address Structure

- The IPv6 socket address is defined by including the <netinet/in.h> header

```
struct in6_addr {
  uint8_t  s6_addr[16];          /* 128-bit IPv6 address */
                                 /* network byte ordered */
};

#define SIN6_LEN      /* required for compile-time tests */

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# */
                                 /* 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 */
};
```

New Generic Socket Address Structure

- A new generic socket address structure was defined as part of the IPv6 sockets API
- This is to overcome some of the shortcomings of the existing struct sockaddr
- The new struct sockaddr_storage is large enough to hold any socket address type supported by the system

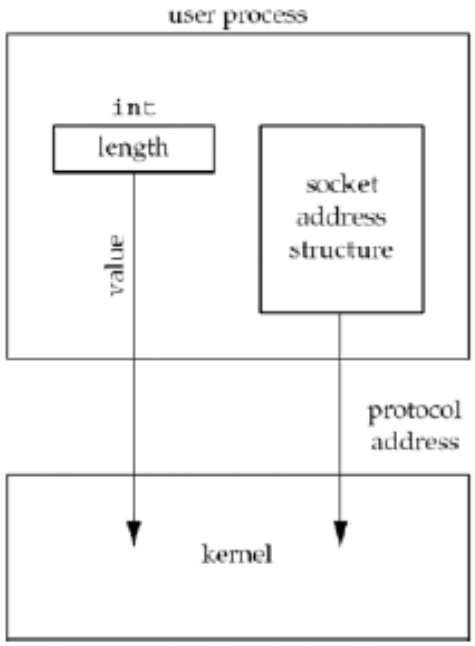
```
struct sockaddr_storage {
  uint8_t      ss_len;       /* length of this struct (implementation dependent)
  sa_family_t  ss_family;    /* address family: AF_xxx value */
  /* implementation-dependent elements to provide:
   * a) alignment sufficient to fulfill the alignment requirements of
   *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

Value-Result Arguments

- We mentioned that when a socket address structure is passed to any socket function, it is always passed by reference
  - That is, a pointer to the structure is passed
- The length of the structure is also passed as an argument
  - But the way in which the length is passed depends on which direction the structure is being passed:
    - from the process to the kernel
      - bind, connect, and sendto, pass a socket address structure from the process to the kernel
    - or vice versa
      - accept, recvfrom, getsockname, and getpeername, pass a socket address structure from the kernel to the process

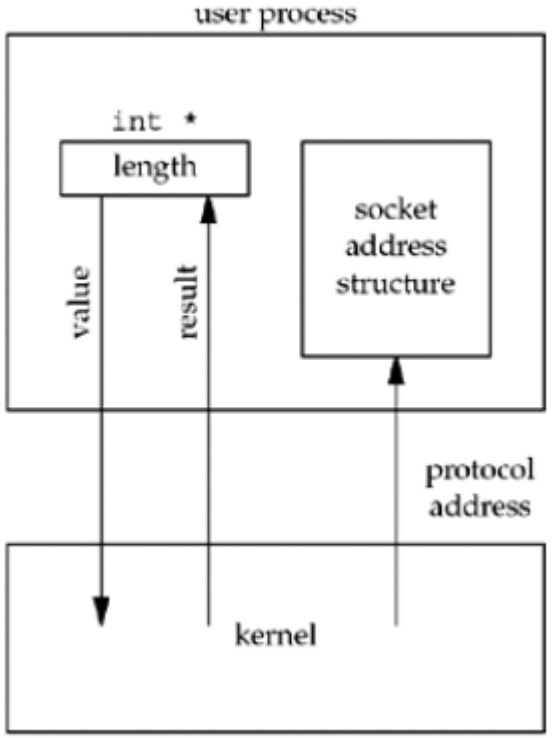
Socket address structure passed from process to kernel

```
#define    SA          struct sockaddr
struct sockaddr_in serv;

/* fill in serv{} */
connect (sockfd, (SA *) &serv, sizeof(serv));
```



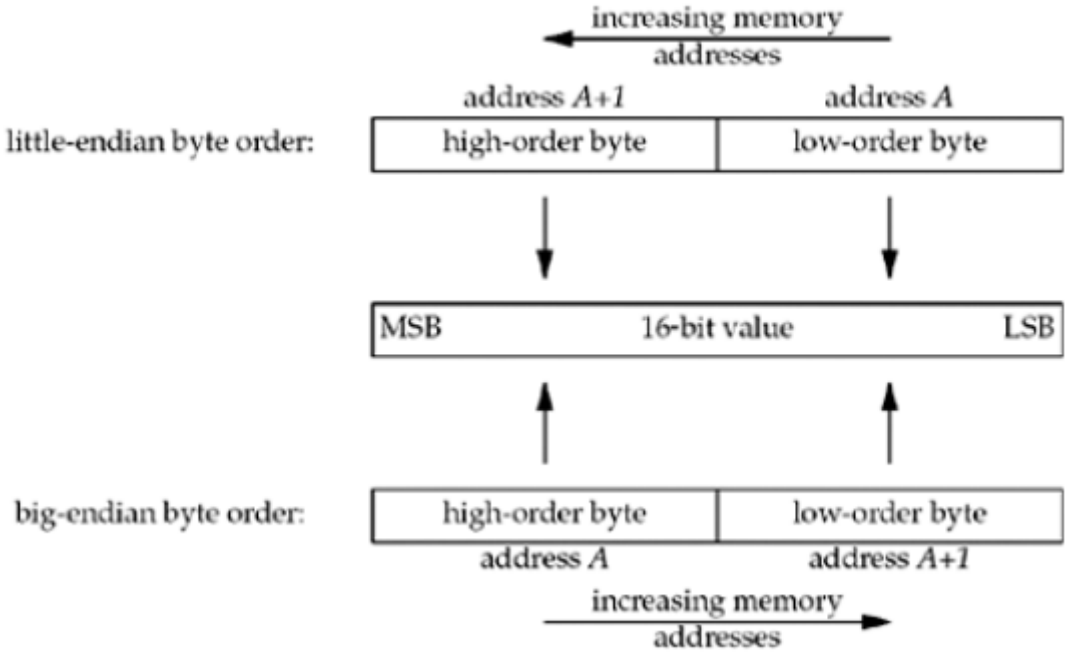
Socket address structure passed from kernel to process

```
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 */
```



Byte Ordering Functions

- Consider a 16-bit integer that is made up of 2 bytes
- There are two ways to store the two bytes in memory:
  - with the low-order byte at the starting address, known as *little-endian* byteorder
  - with the high-order byte at the starting address, known as *big-endian* byte order



Byte Manipulation Functions

- There are two groups of functions that operate on multibyte fields, without interpreting the data, and without assuming that the data is a null-terminated C string
- We need these types of functions when dealing with socket address structures because we need to manipulate fields such as IP addresses
- The first group of functions, whose names begin with b (for byte), are from 4.2BSD and are still provided by almost any system that supports the socket functions
- The second group of functions, whose names begin with mem (for memory), are from the ANSI C standard and are provided with any system that supports an ANSI C library

Byte Manipulation Functions

```
#include <strings.h>
void bzero(void *dest,size_t nbytes);
void bcopy(const void *src,void *dest,size_t nbytes);
int bcmp(const void *ptr1,const void *ptr2,size_t nbytes);
```

Returns: 0 if equal, nonzero if unequal

```
#include <string.h>
void *memset(void *dest,int c,size_t len);
void *memcpy(void *dest,const void *src,size_t nbytes);
int memcmp(const void *ptr1,const void *ptr2,size_t nbytes);
```

Returns: 0 if equal, <0 or >0 if unequal (see text)

inet_aton, inet_addr, and inet_ntoa Functions

- inet_aton, inet_ntoa, and inet_addr convert an IPv4 address from a dotted-decimal string (e.g., "206.168.112.96") to its 32-bit network byte ordered binary value

`#include <arpa/inet.h>`
`int inet_aton(const char *strptr,struct in_addr *addrptr);`
Returns: 1 if string was valid, 0 on error
`in_addr_t inet_addr(const char *strptr);`
Returns: 32-bit binary network byte ordered IPv4 address; `INADDR_NONE` if error
`char *inet_ntoa(struct in_addrinaddr);`
Returns: pointer to dotted-decimal string

int inet_aton(const char *_strptr_, struct in_addr *_addrptr_);

- inet_aton, converts the C character string pointed to by *strptr* into its 32- bit binary network byte ordered value, which is stored through the pointer *addrptr*
- If successful, 1 is returned; otherwise, 0 is returned

in_addr_t inet_addr(const char *_strptr_);

- inet_addr does the same conversion, returning the 32-bit binary network byte ordered value as the return value

char *inet_ntoa(struct in_addr *inaddr*);

- The inet_ntoa function converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal string

inet_pton and inet_ntop Functions

- These two functions work with both IPv4 and IPv6 addresses
- The letters "p" and "n" stand for *presentation (*ASCII string) and *numeric (*binary value)

```
#include <arpa/inet.h>
int inet_pton(int family, const char *strptr, void *addrptr);
```

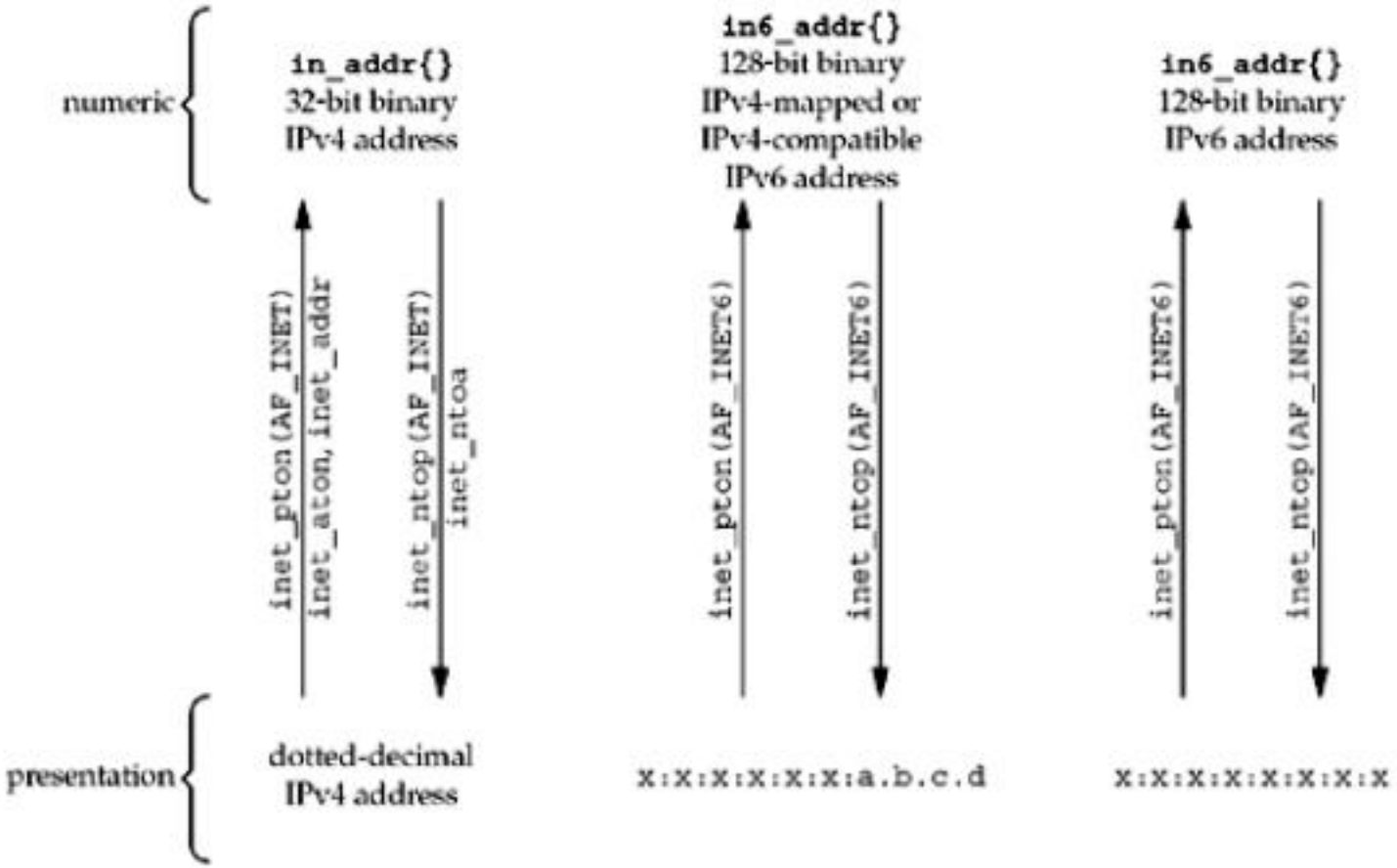
Returns: 1 if OK, 0 if input not a valid presentation format, -1 on error

```
const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len);
```

Returns: pointer to result if OK, NULL on error

- The *family* argument for both functions is either AF_INET or AF_INET6

Summary of address conversion functions



Elementary TCP Sockets

Socket functions for elementary TCP client/server