523454

Computer Network Programming

Overview of TCP connections

Dr. Parin Sornlertlamvanich,

parin.s@sut.ac.th

Socket Address Structures

IPv4 Socket Address Structure

```
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 */
};
```

Socket Address Structures

- Generic Socket Address Structure

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

```
int bind(int, struct sockaddr *, socklen_t);
struct sockaddr_in  serv;           /* IPv4 socket address structure */

/* fill in serv{} */

bind(sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

Socket Address Structures

- IPv6 Socket Address Structure

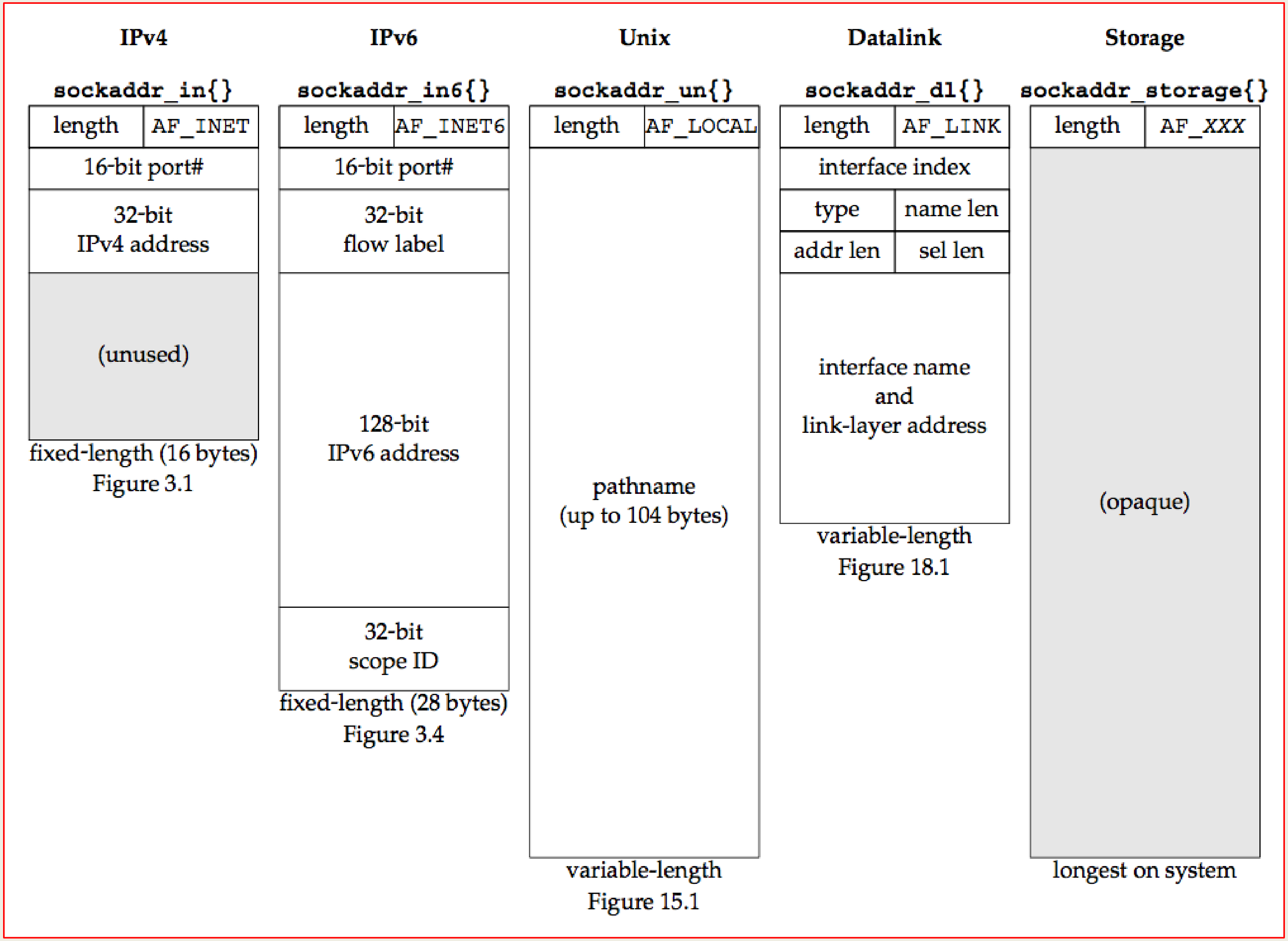
```
struct in6_addr {
  uint8_t  s6_addr[16];                        /* 128-bit IPv6 address */
                                               /* network byte ordered */
};


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 */
};
```

Socket Address Structures

- New Generic Socket Address Structure

```
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.
   */
};
```

Comparison of Socket Address Structures



IPv4

sockaddr_in{}

length | AF_INET

16-bit port#

32-bit IPv4 address

(unused)

fixed-length (16 bytes)

Figure 3.1

IPv6

sockaddr_in6{}

length | AF_INET6

16-bit port#

32-bit flow label

128-bit IPv6 address

32-bit scope ID

fixed-length (28 bytes)

Figure 3.4

Unix

sockaddr_un{}

length | AF_LOCAL

pathname (up to 104 bytes)

variable-length

Figure 15.1

Datalink

sockaddr_dl{}

length | AF_LINK

interface index

type | name len

addr len | sel len

interface name and link-layer address

variable-length

Figure 18.1

Storage

sockaddr_storage{}

length | AF_XXX

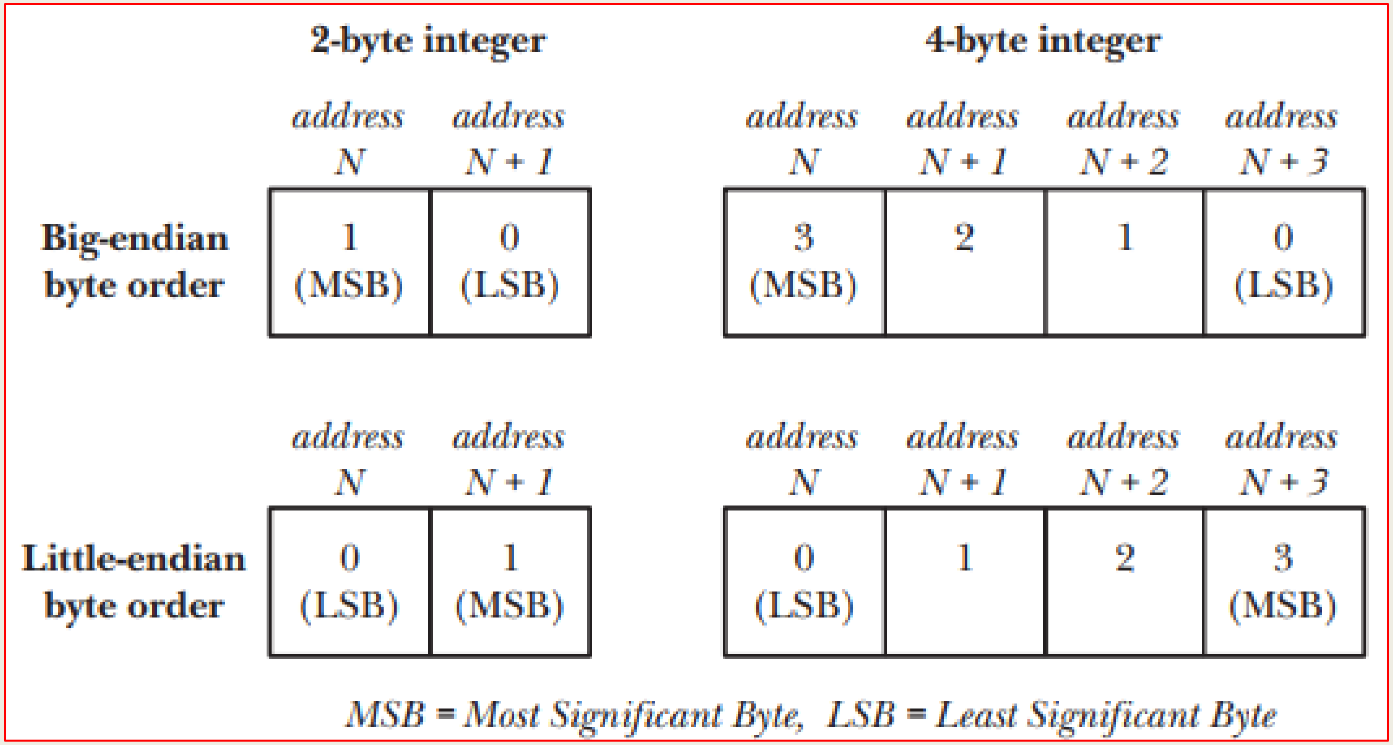
(opaque)

longest on system

Byte Ordering Functions

- For a 16-bit integer that is made up of 2 bytes
  - there are two ways to store the two bytes in memory

1. Little-endian order: low-order byte is at the starting address.
2. Big-endian order: high-order byte is at the starting address.



2-byte integer

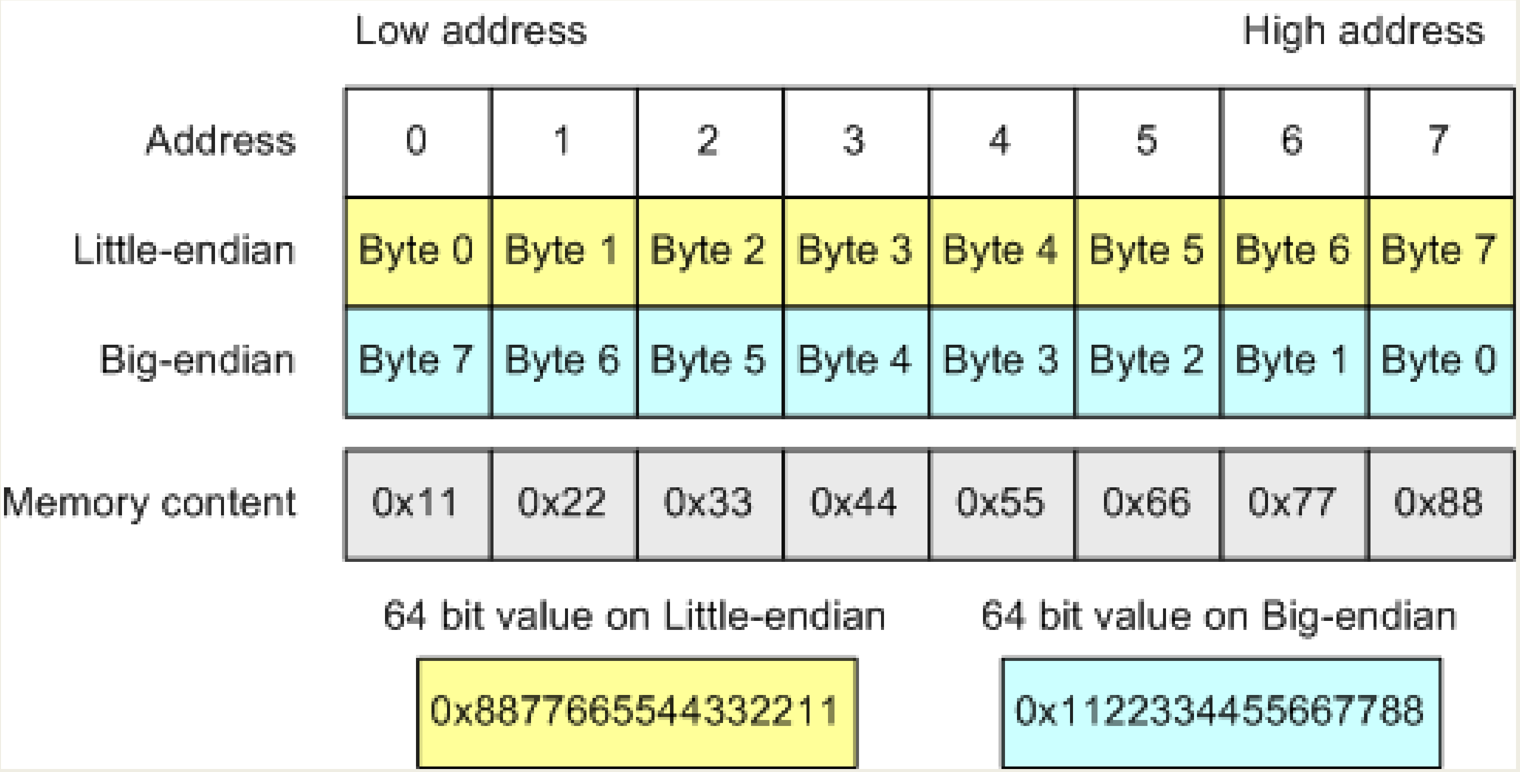
	address N	*address N + 1*
Big-endian byte order	1 (MSB)	0 (LSB)
Little-endian byte order	0 (LSB)	1 (MSB)

4-byte integer

	address N	*address N + 1*	*address N + 2*	*address N + 3*
Big-endian byte order	3 (MSB)	2	1	0 (LSB)
Little-endian byte order	0 (LSB)	1	2	3 (MSB)

MSB = Most Significant Byte, LSB = Least Significant Byte

Byte Ordering Functions



Low address High address

Address	0	1	2	3	4	5	6	7
Little-endian	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Big-endian	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
Memory content	0x11	0x22	0x33	0x44	0x55	0x66	0x77	0x88

64 bit value on Little-endian

0x8877665544332211

64 bit value on Big-endian

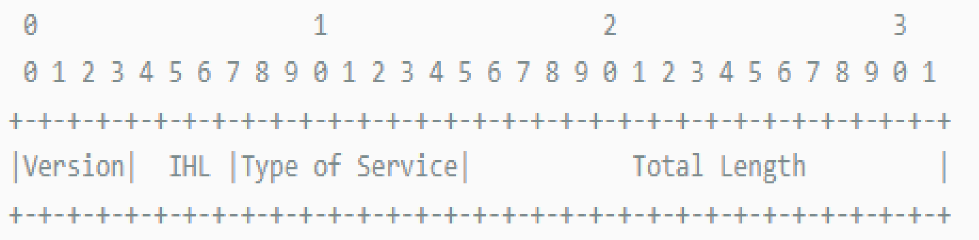
0x1122334455667788

Byte Ordering Functions

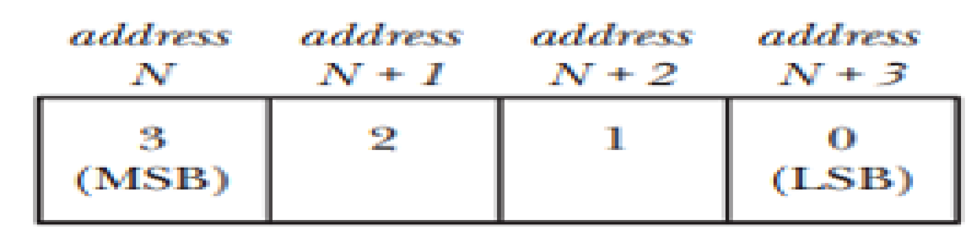
- The terms "little-endian" and "big-endian" indicate which end of the multibyte value, the little end or the big end, is stored at the starting address of the value.
- Host byte order refer to the byte ordering used by a given system.
  - Little-endian byte ordering
- Networking protocols must specify a Network byte order.
  - The Internet protocols use big-endian byte ordering for these multibyte integers

Byte Ordering Functions

- Bit ordering is an important convention in Internet standards, such as the first 32 bits of the IPv4 header from RFC 791:



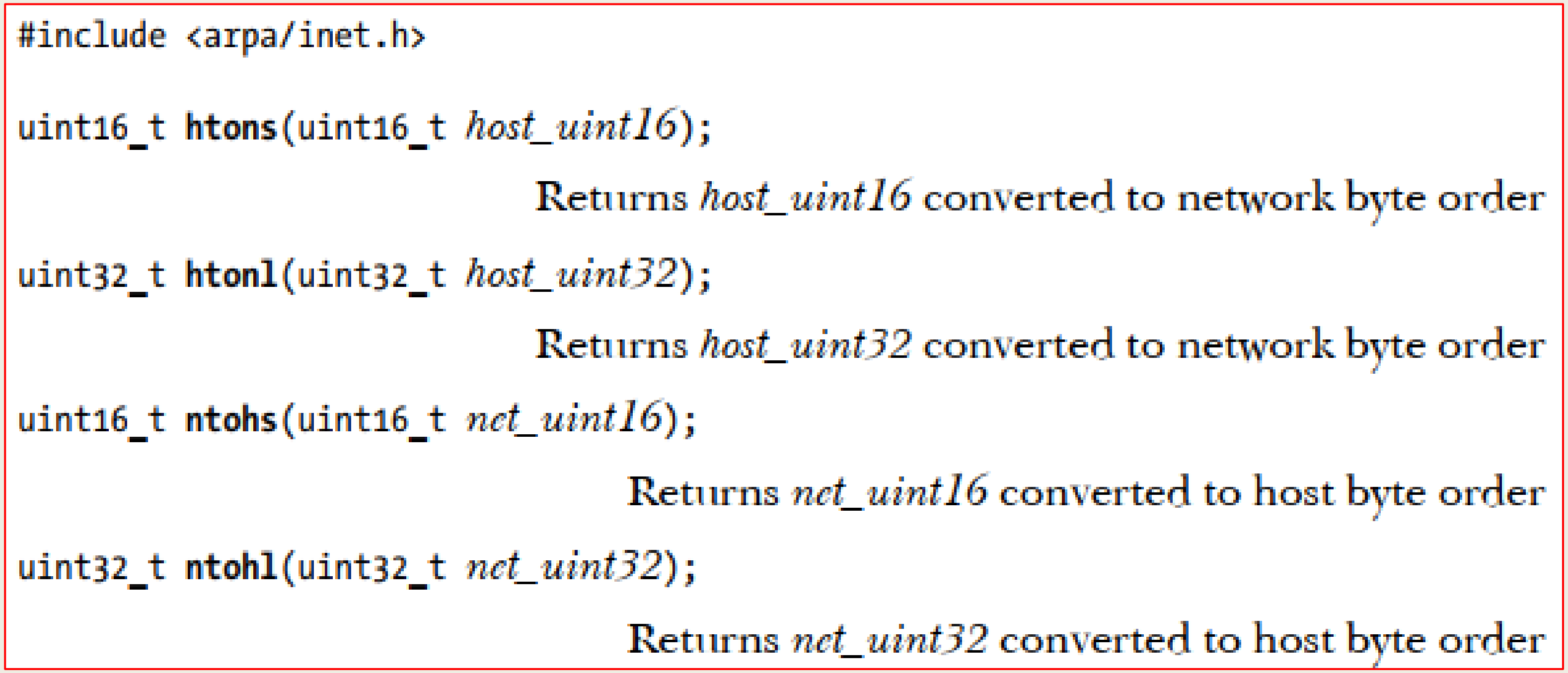
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```



address *N*	address *N + 1*	address *N + 2*	address *N + 3*
3 (MSB)	2	1	0 (LSB)

- the leftmost bit is the most significant. However, the numbering starts with zero assigned to the most significant bit.

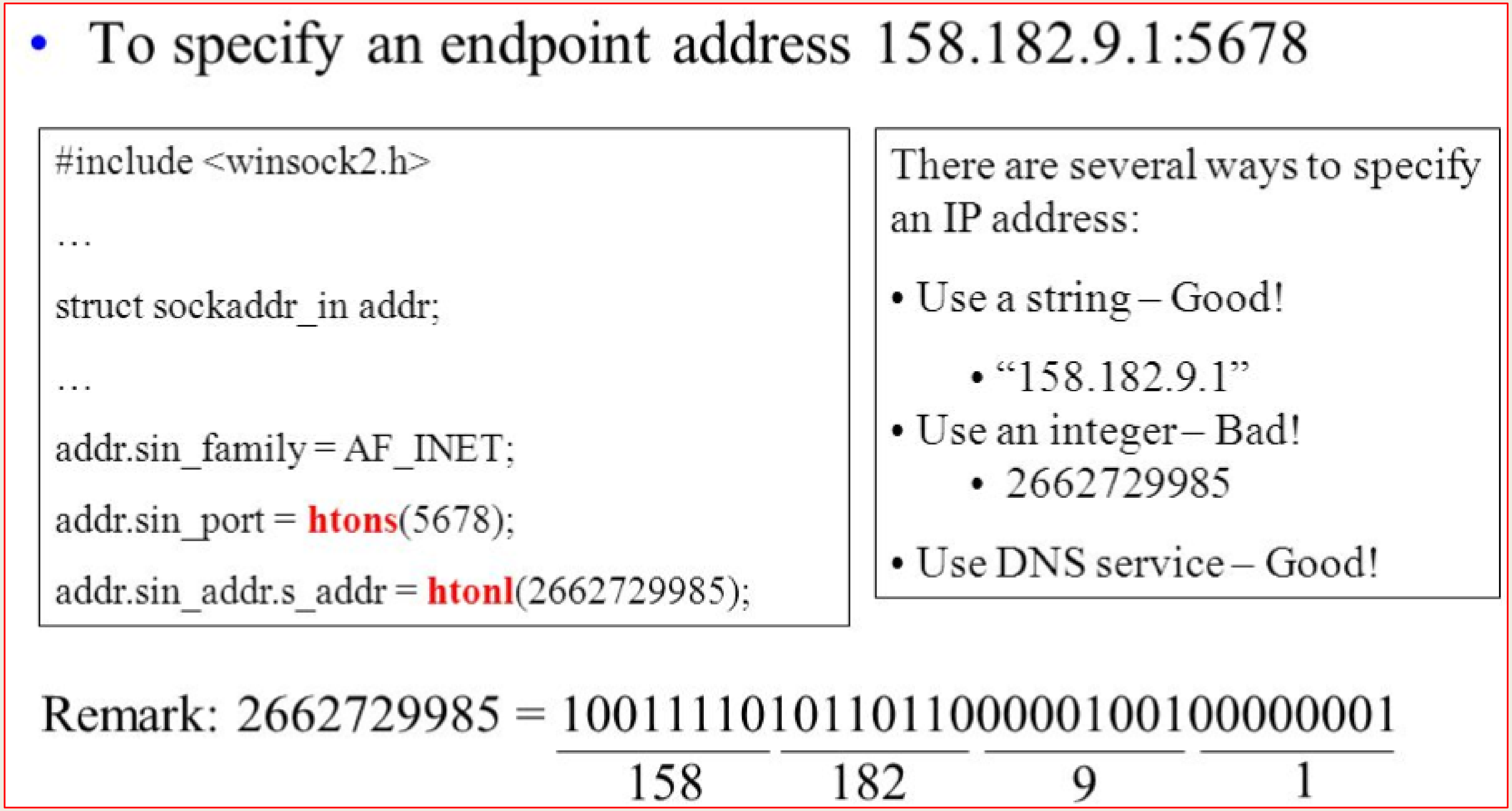
Byte Ordering Functions



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

uint16_t htons(uint16_t host_uint16);
                          Returns host_uint16 converted to network byte order
uint32_t htonl(uint32_t host_uint32);
                          Returns host_uint32 converted to network byte order
uint16_t ntohs(uint16_t net_uint16);
                              Returns net_uint16 converted to host byte order
uint32_t ntohl(uint32_t net_uint32);
                              Returns net_uint32 converted to host byte order
```

- h stands for host
- n stands for network
- s stands for short (16-bit value, e.g. TCP or UDP port number)
- l stands for long (32-bit value, e.g. IPv4 address)

Byte Ordering Functions



- To specify an endpoint address 158.182.9.1:5678

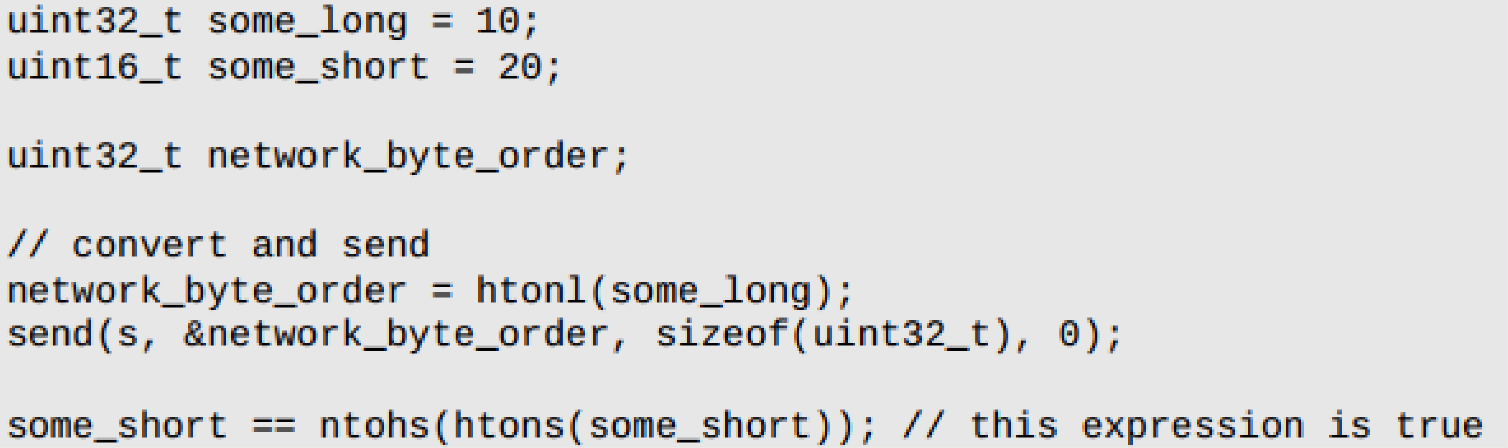
```
#include <winsock2.h>

…

struct sockaddr_in addr;

…

addr.sin_family = AF_INET;

addr.sin_port = htons(5678);

addr.sin_addr.s_addr = htonl(2662729985);
```

There are several ways to specify an IP address:

- Use a string – Good!
  - "158.182.9.1"
- Use an integer – Bad!
  - 2662729985
- Use DNS service – Good!

Remark: $2662729985 = \underbrace{10011110}_{158}\underbrace{10110110}_{182}\underbrace{00001001}_{9}\underbrace{00000001}_{1}$

Byte Ordering Functions

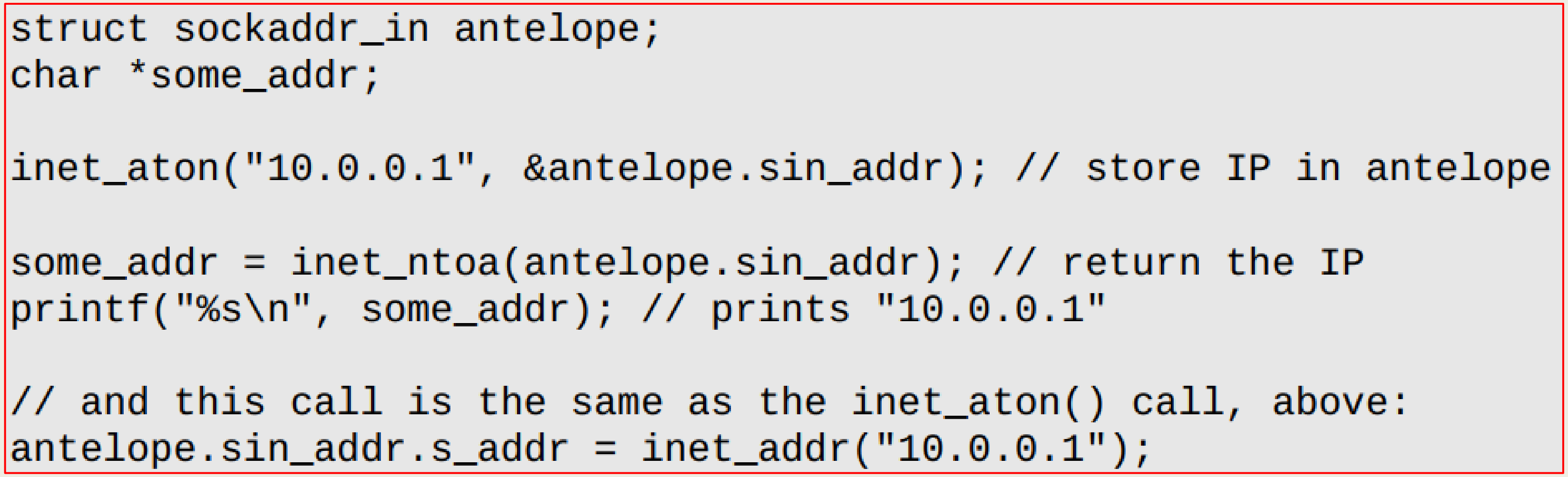


```
uint32_t some_long = 10;
uint16_t some_short = 20;

uint32_t network_byte_order;

// convert and send
network_byte_order = htonl(some_long);
send(s, &network_byte_order, sizeof(uint32_t), 0);

some_short == ntohs(htons(some_short)); // this expression is true
```

inet_aton and inet_ntoa Functions

- These functions convert Internet addresses between
  - ASCII strings - what humans prefer to use
  - Network byte ordered binary values - values that are stored in socket address structures
- inet_aton:
  - int inet_aton(const char *strptr, struct in_addr *addrptr);
  - converting from a dots-and-numbers string into a network address in a struct in_addr
- inet_ntoa:
  - char *inet_ntoa(struct in_addr inaddr);
  - converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal (dots-and-numbers format) string.

inet_aton and inet_ntoa Functions



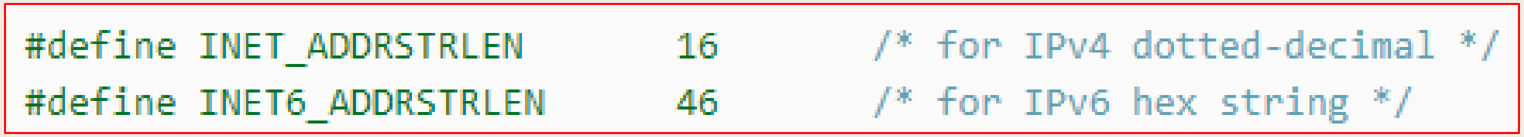
```
struct sockaddr_in antelope;
char *some_addr;

inet_aton("10.0.0.1", &antelope.sin_addr); // store IP in antelope

some_addr = inet_ntoa(antelope.sin_addr); // return the IP
printf("%s\n", some_addr); // prints "10.0.0.1"

// and this call is the same as the inet_aton() call, above:
antelope.sin_addr.s_addr = inet_addr("10.0.0.1");
```

inet_pton and inet_ntop Functions

- These two functions are new with IPv6 and work with both IPv4 and IPv6 addresses

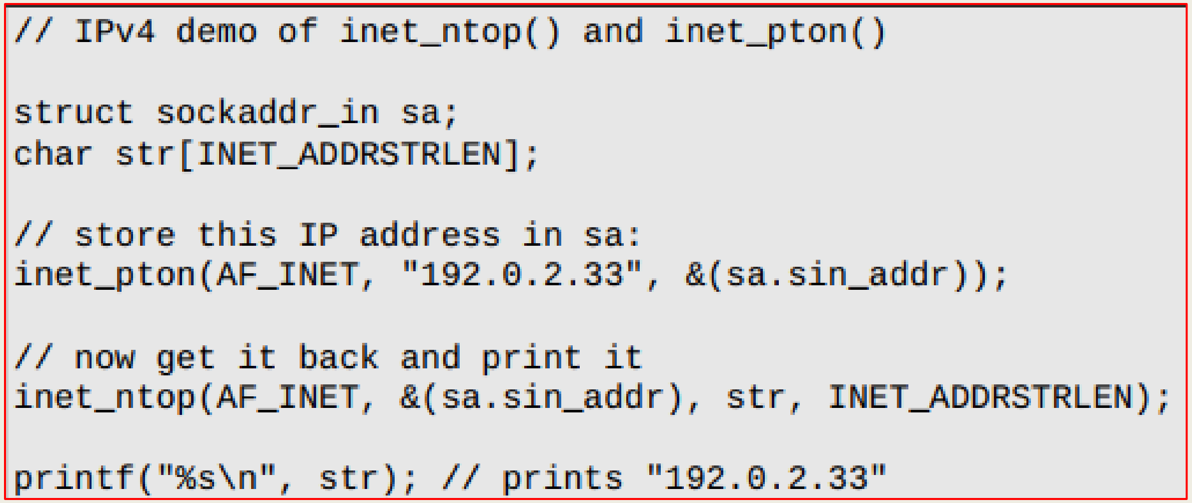
- The letters "p" and "n" stand for presentation and numeric
  - The presentation format for an address is often an ASCII string
  - the numeric format is the binary value that goes into a socket address structure

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

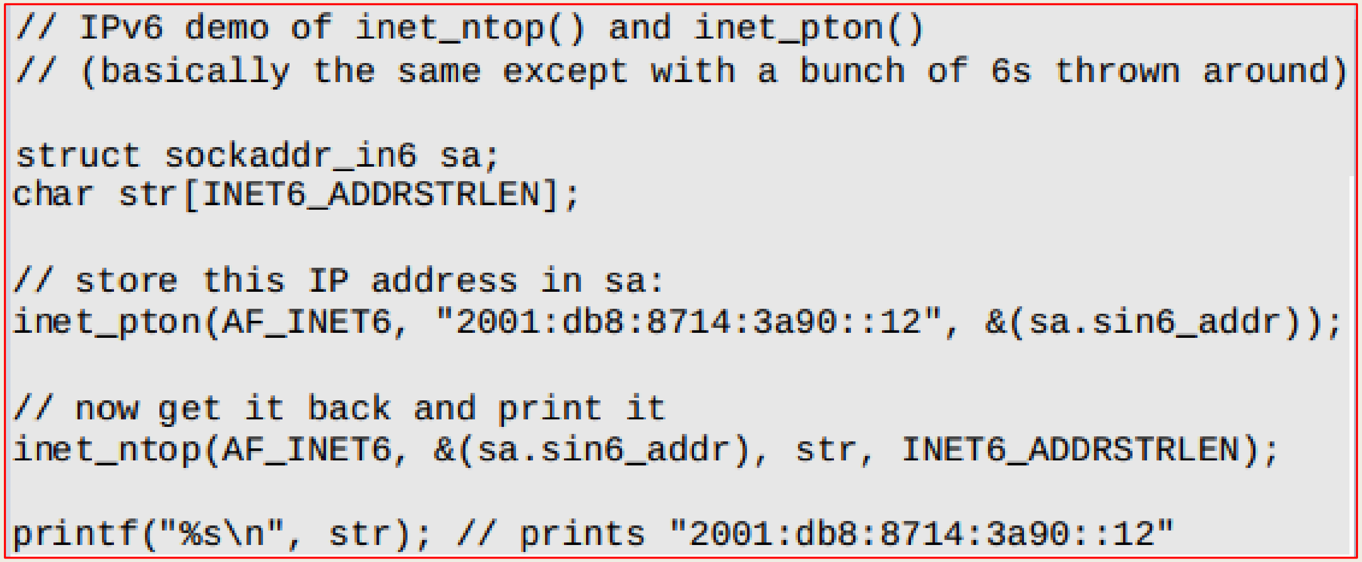


```
#define INET_ADDRSTRLEN      16       /* for IPv4 dotted-decimal */
#define INET6_ADDRSTRLEN     46       /* for IPv6 hex string */
```

inet_pton and inet_ntop Functions

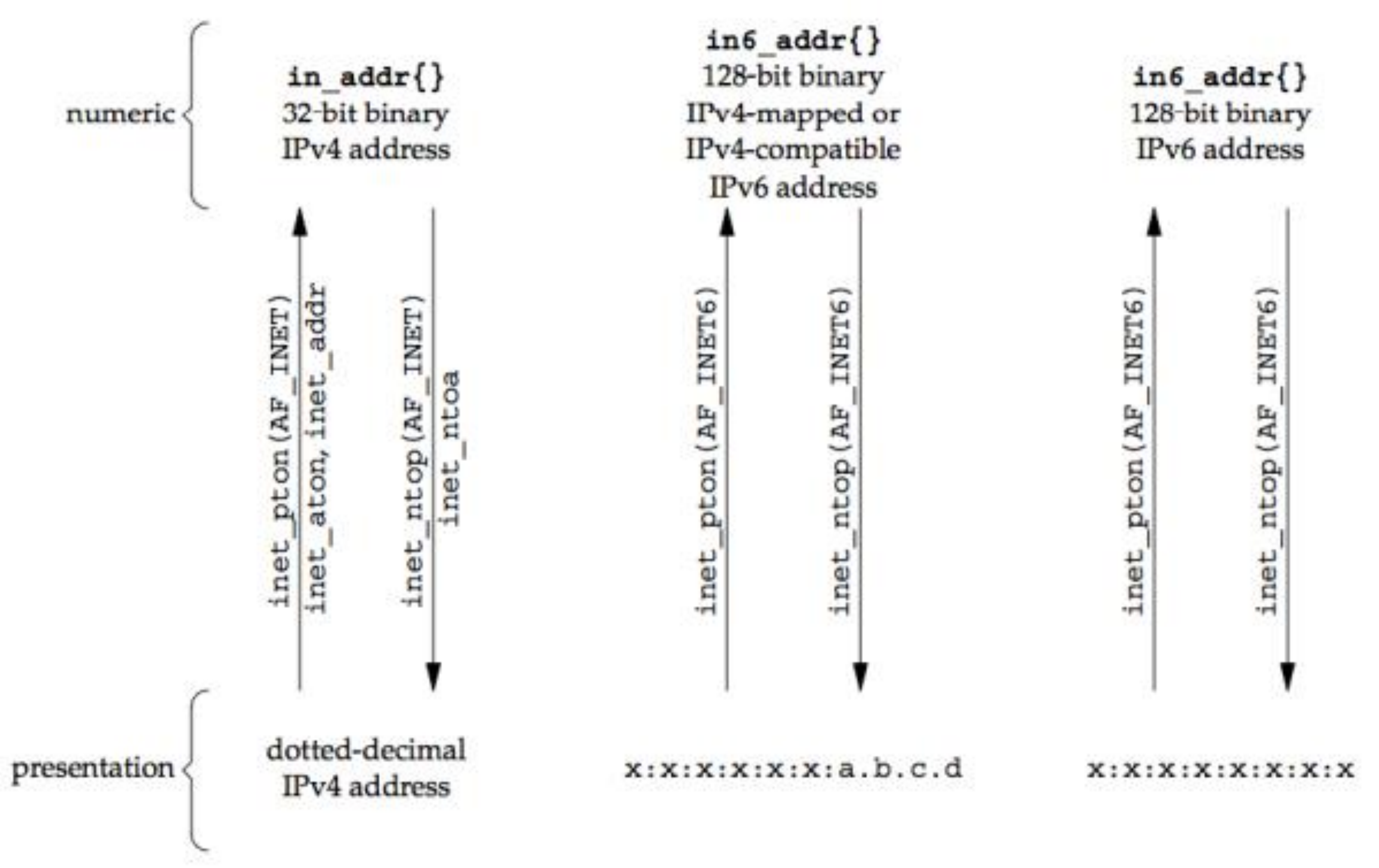


```
// IPv4 demo of inet_ntop() and inet_pton()

struct sockaddr_in sa;
char str[INET_ADDRSTRLEN];

// store this IP address in sa:
inet_pton(AF_INET, "192.0.2.33", &(sa.sin_addr));

// now get it back and print it
inet_ntop(AF_INET, &(sa.sin_addr), str, INET_ADDRSTRLEN);

printf("%s\n", str); // prints "192.0.2.33"
```

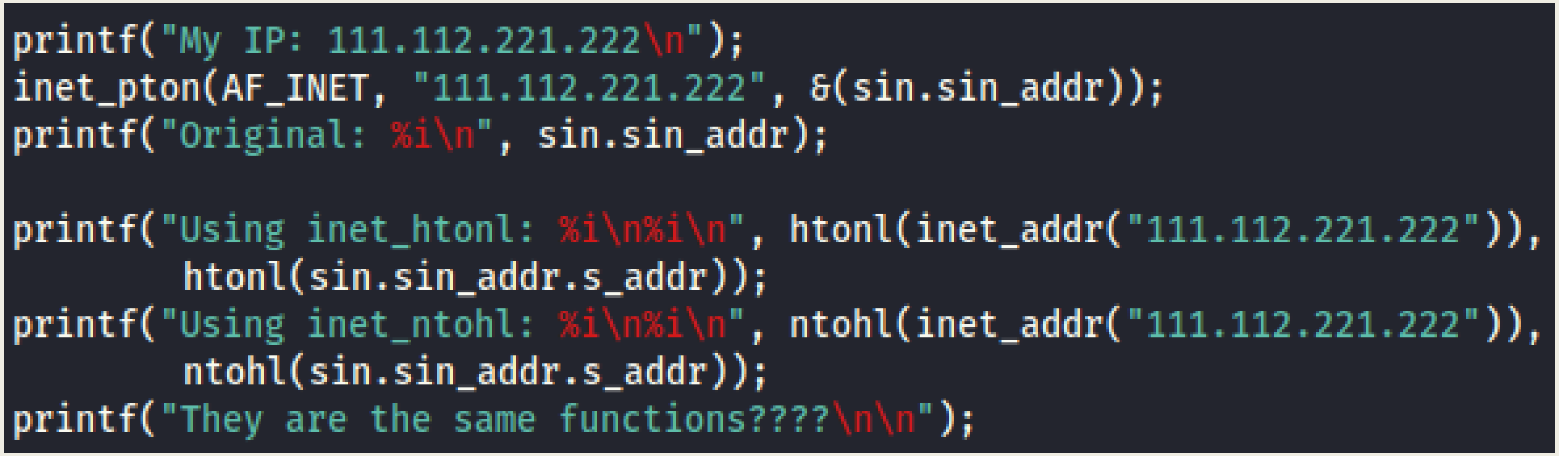


```
// IPv6 demo of inet_ntop() and inet_pton()
// (basically the same except with a bunch of 6s thrown around)

struct sockaddr_in6 sa;
char str[INET6_ADDRSTRLEN];

// store this IP address in sa:
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &(sa.sin6_addr));

// now get it back and print it
inet_ntop(AF_INET6, &(sa.sin6_addr), str, INET6_ADDRSTRLEN);

printf("%s\n", str); // prints "2001:db8:8714:3a90::12"
```

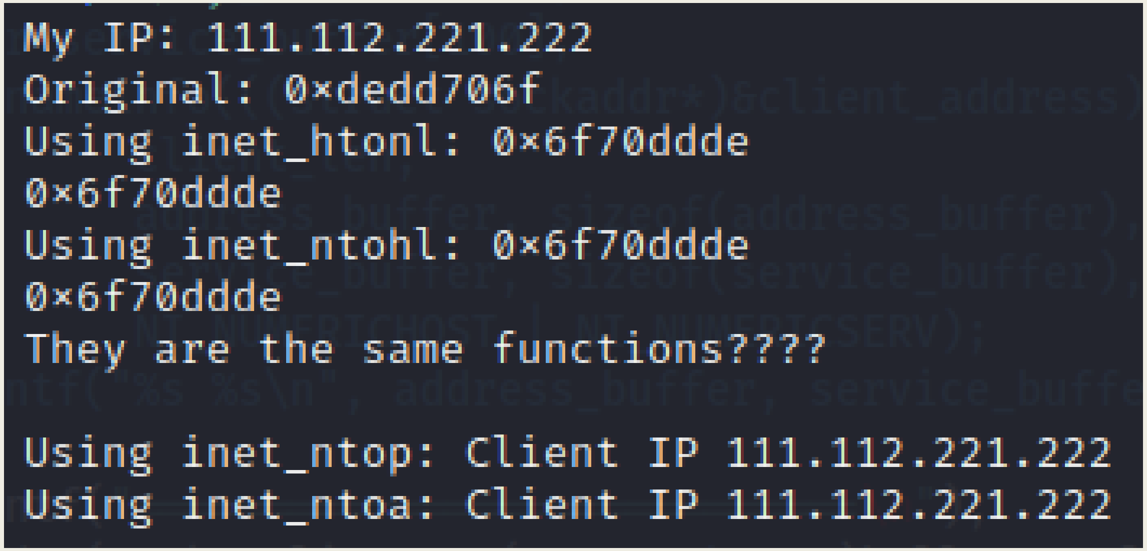
4 functions on address conversion



byte_order.c



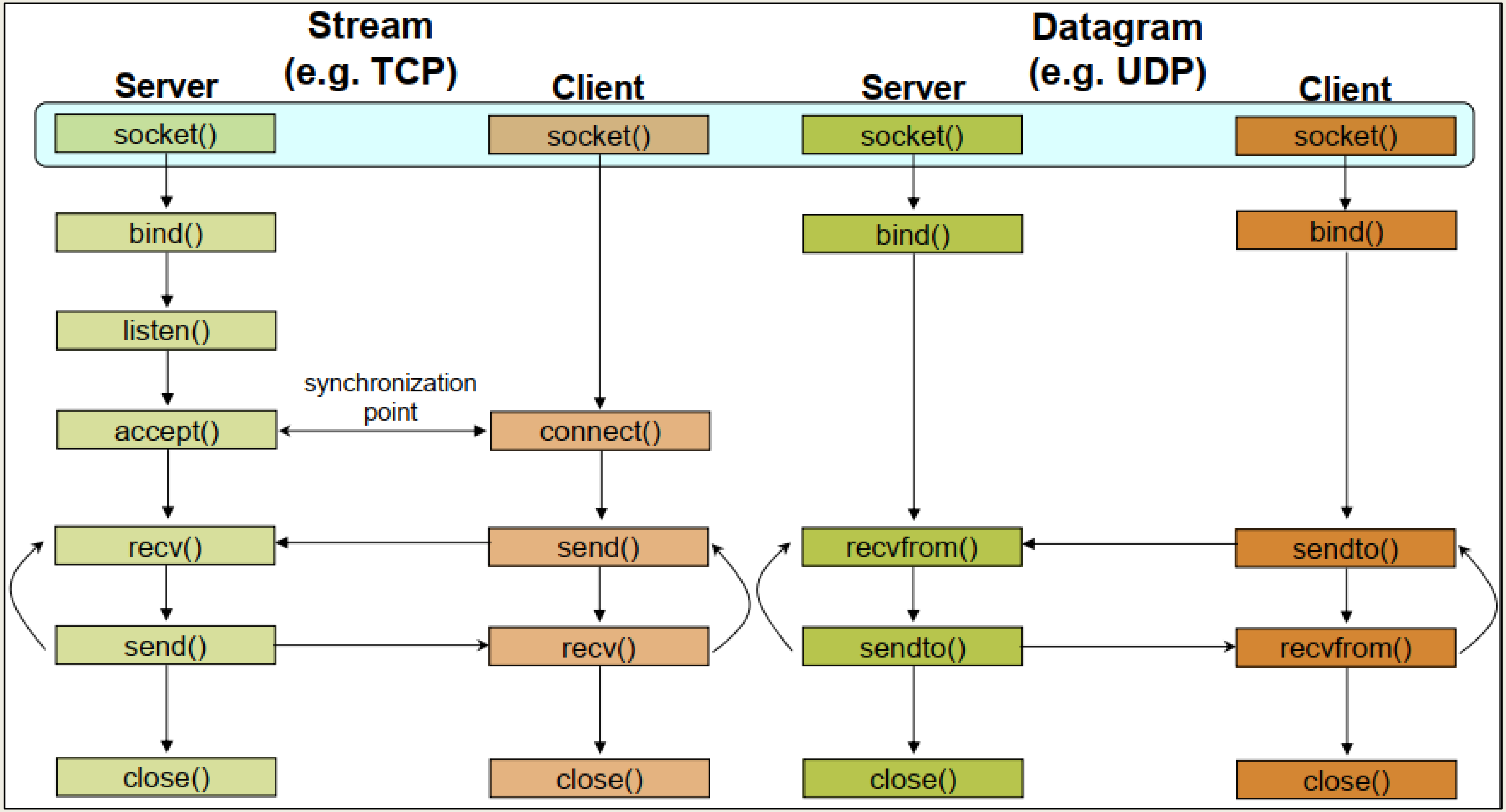
```c
printf("My IP: 111.112.221.222\n");
inet_pton(AF_INET, "111.112.221.222", &(sin.sin_addr));
printf("Original: %i\n", sin.sin_addr);

printf("Using inet_htonl: %i\n%i\n", htonl(inet_addr("111.112.221.222")),
        htonl(sin.sin_addr.s_addr));
printf("Using inet_ntohl: %i\n%i\n", ntohl(inet_addr("111.112.221.222")),
        ntohl(sin.sin_addr.s_addr));
printf("They are the same functions????\n\n");
```

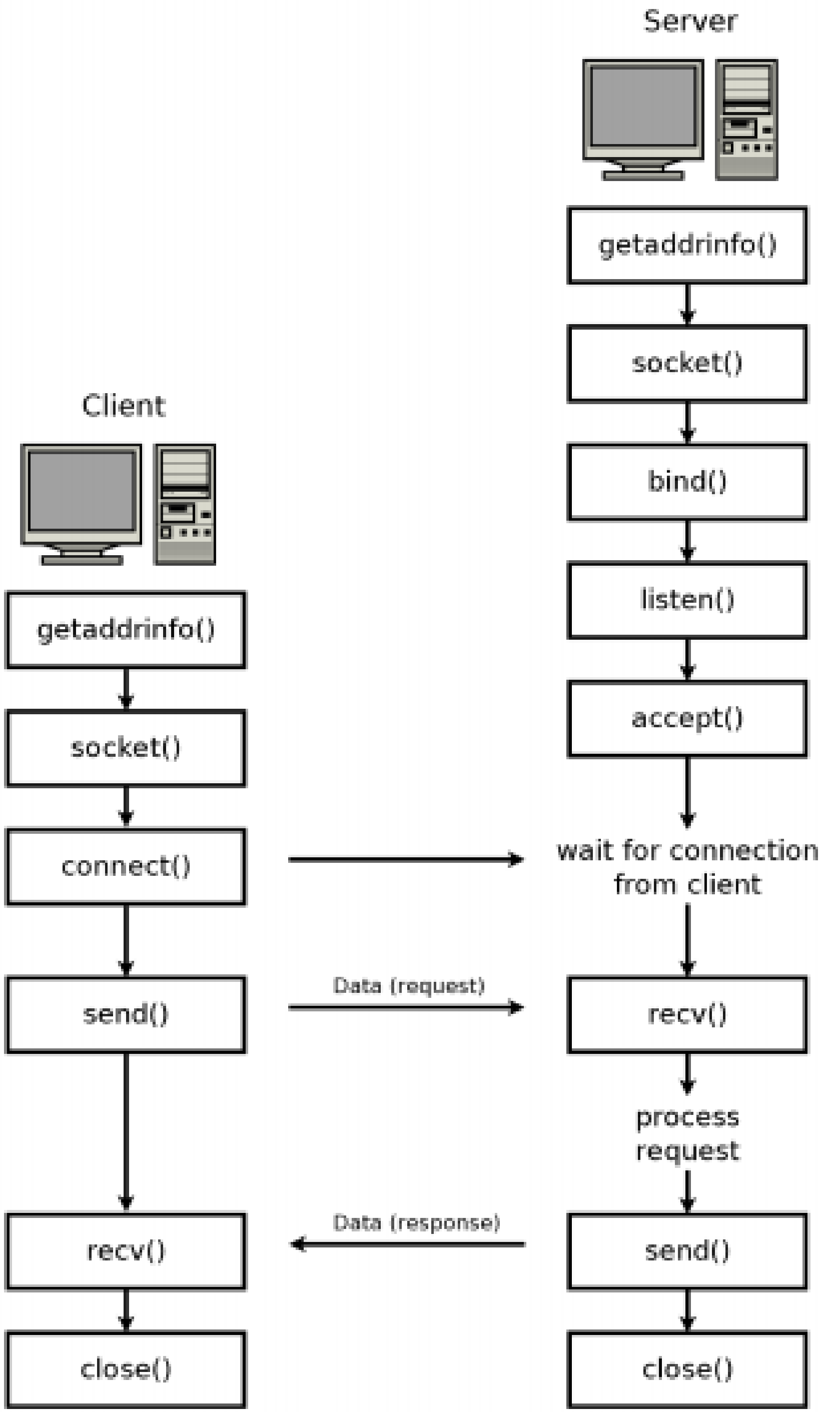


```
My IP: 111.112.221.222
Original: 0×dedd706f
Using inet_htonl: 0×6f70ddde
0×6f70ddde
Using inet_ntohl: 0×6f70ddde
0×6f70ddde
They are the same functions????

Using inet_ntop: Client IP 111.112.221.222
Using inet_ntoa: Client IP 111.112.221.222
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>

int main() {
struct sockaddr_in sin;
struct sockaddr_storage client_address;
struct sockaddr_in *sin_storage;
char ipv4[INET_ADDRSTRLEN]
```

Client - Server Communication



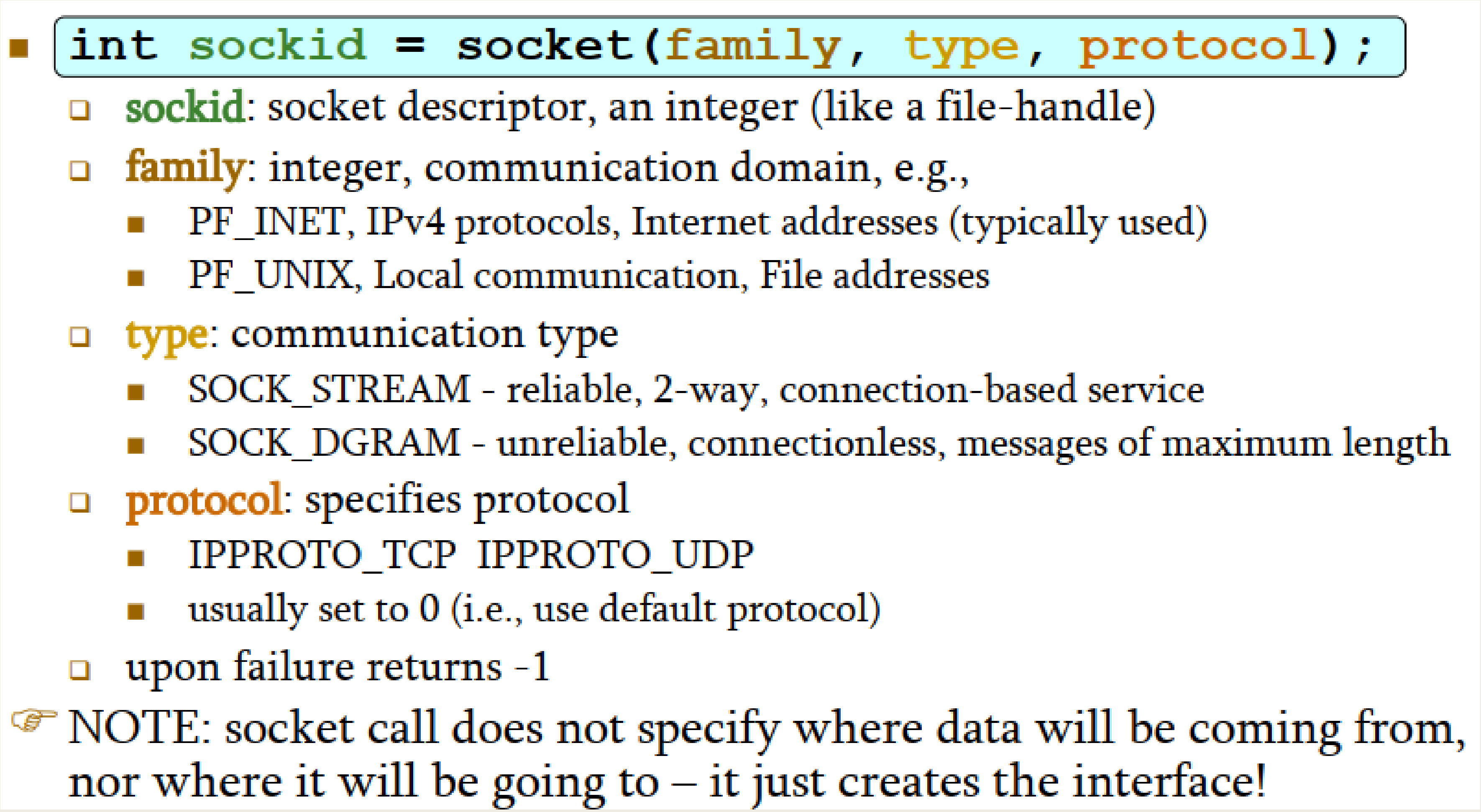


TCP: client & server

1. The server then creates the socket with a call to socket()
2. The socket must be bound to the listening IP address and port - call to bind()
3. then calls listen(), which puts the socket in a state where it listens for new connections
4. then call accept(), which will wait until a client establishes a connection to the server
5. When the new connection has been established, accept() returns a *new* socket
6. This *new* socket can be used to exchange data with the client using send() and recv()

**Meanwhile, the first socket remains listening for new connections, and repeated calls to accept()

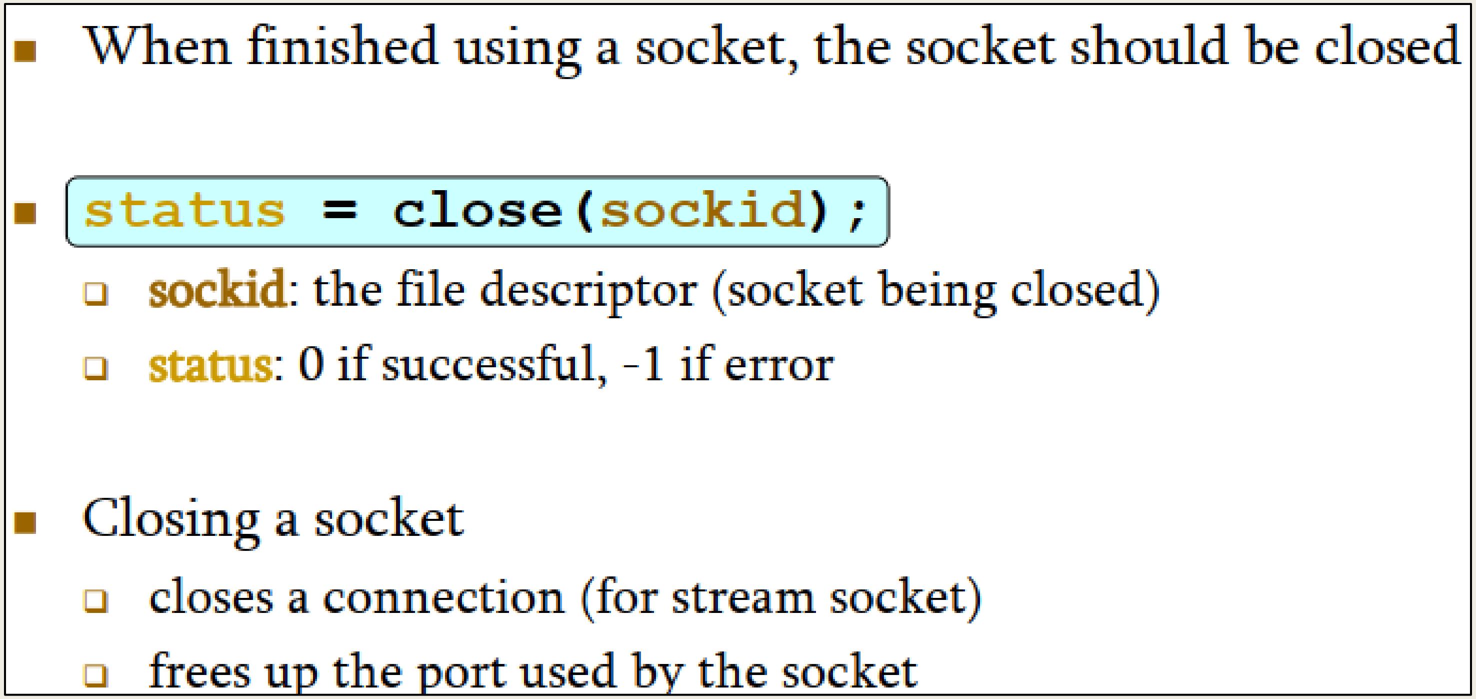
socket()



- `int sockid = socket(family, type, protocol);`
  - sockid: socket descriptor, an integer (like a file-handle)
  - family: integer, communication domain, e.g.,
    - PF_INET, IPv4 protocols, Internet addresses (typically used)
    - PF_UNIX, Local communication, File addresses
  - type: communication type
    - SOCK_STREAM - reliable, 2-way, connection-based service
    - SOCK_DGRAM - unreliable, connectionless, messages of maximum length
  - protocol: specifies protocol
    - IPPROTO_TCP IPPROTO_UDP
    - usually set to 0 (i.e., use default protocol)
  - upon failure returns -1

☞ NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

close()



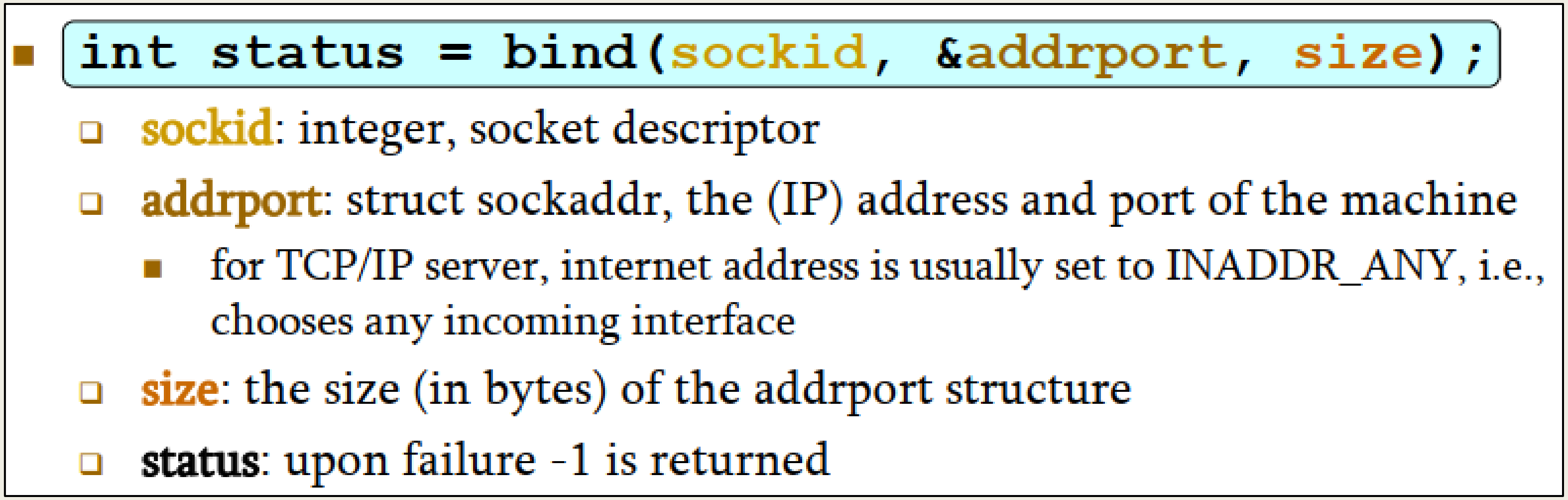
- When finished using a socket, the socket should be closed

- `status = close(sockid);`
    - sockid: the file descriptor (socket being closed)
    - status: 0 if successful, -1 if error

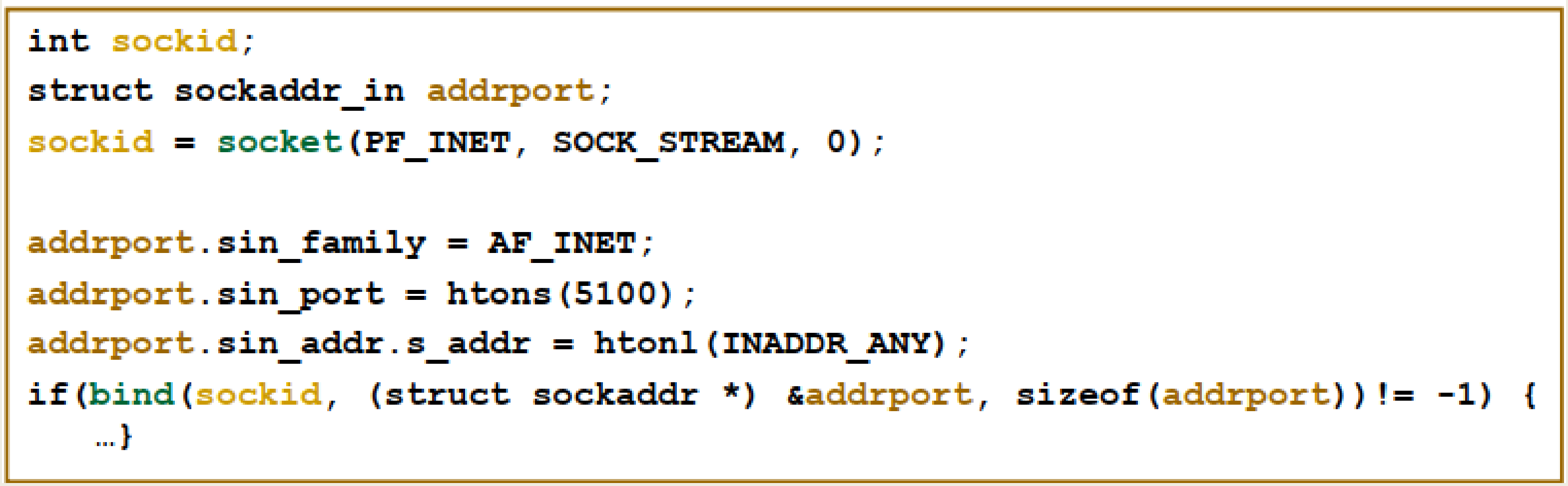
- Closing a socket
    - closes a connection (for stream socket)
    - frees up the port used by the socket

bind()

- associates and reserves a port for use by the socket



- `int status = bind(sockid, &addrport, size);`
    - sockid: integer, socket descriptor
    - addrport: struct sockaddr, the (IP) address and port of the machine
        - for TCP/IP server, internet address is usually set to INADDR_ANY, i.e., chooses any incoming interface
    - size: the size (in bytes) of the addrport structure
    - status: upon failure -1 is returned



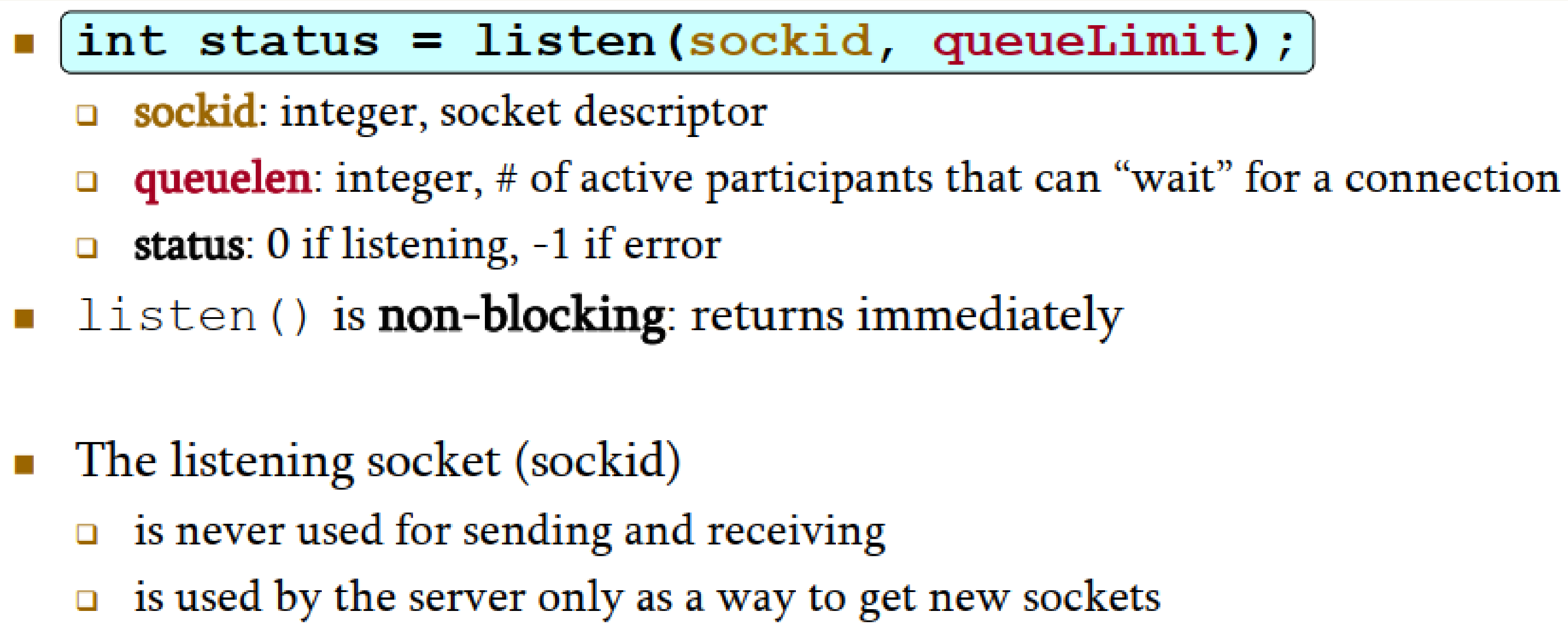
```
int sockid;
struct sockaddr_in addrport;
sockid = socket(PF_INET, SOCK_STREAM, 0);

addrport.sin_family = AF_INET;
addrport.sin_port = htons(5100);
addrport.sin_addr.s_addr = htonl(INADDR_ANY);
if(bind(sockid, (struct sockaddr *) &addrport, sizeof(addrport))!= -1) {
   …}
```

Skipping the bind()

- bind can be skipped for both types of sockets
- Datagram socket:
  - if only sending, no need to bind. The OS finds a port each time the socket sends a packet
  - if receiving, need to bind
- Stream socket:
  - destination determined during connection setup
  - don’t need to know port sending from (during connection setup, receiving end is informed of port)

listen for connections: listen()

- Instructs TCP protocol implementation to listen for connections

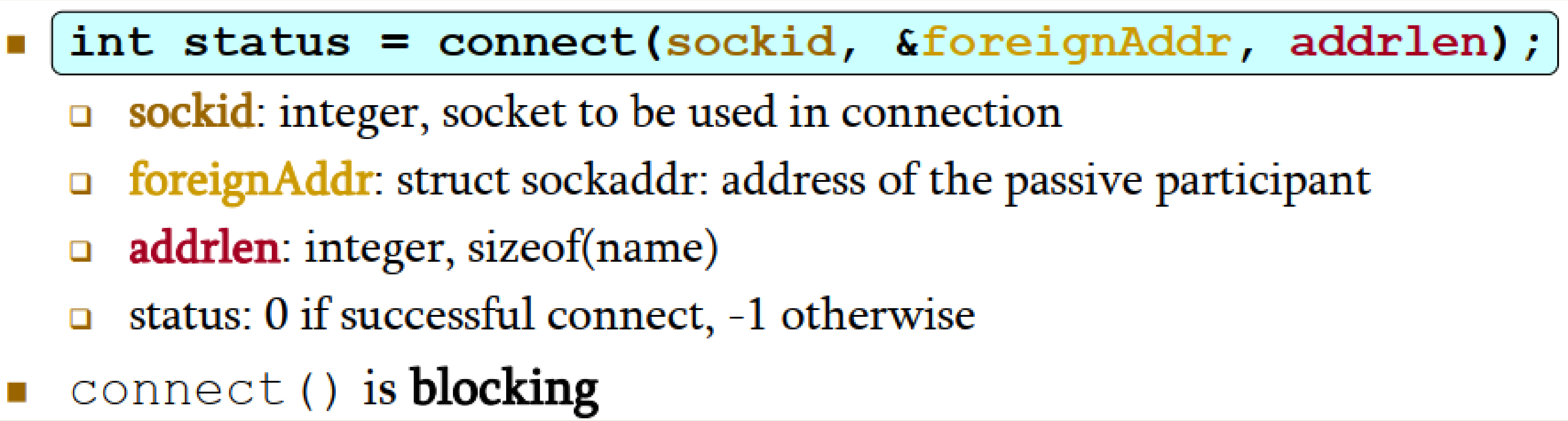


- `int status = listen(sockid, queueLimit);`
  - **sockid**: integer, socket descriptor
  - **queuelen**: integer, # of active participants that can “wait” for a connection
  - **status**: 0 if listening, -1 if error
- `listen()` is **non-blocking**: returns immediately

- The listening socket (sockid)
  - is never used for sending and receiving
  - is used by the server only as a way to get new sockets

Establish Connection: Connect()

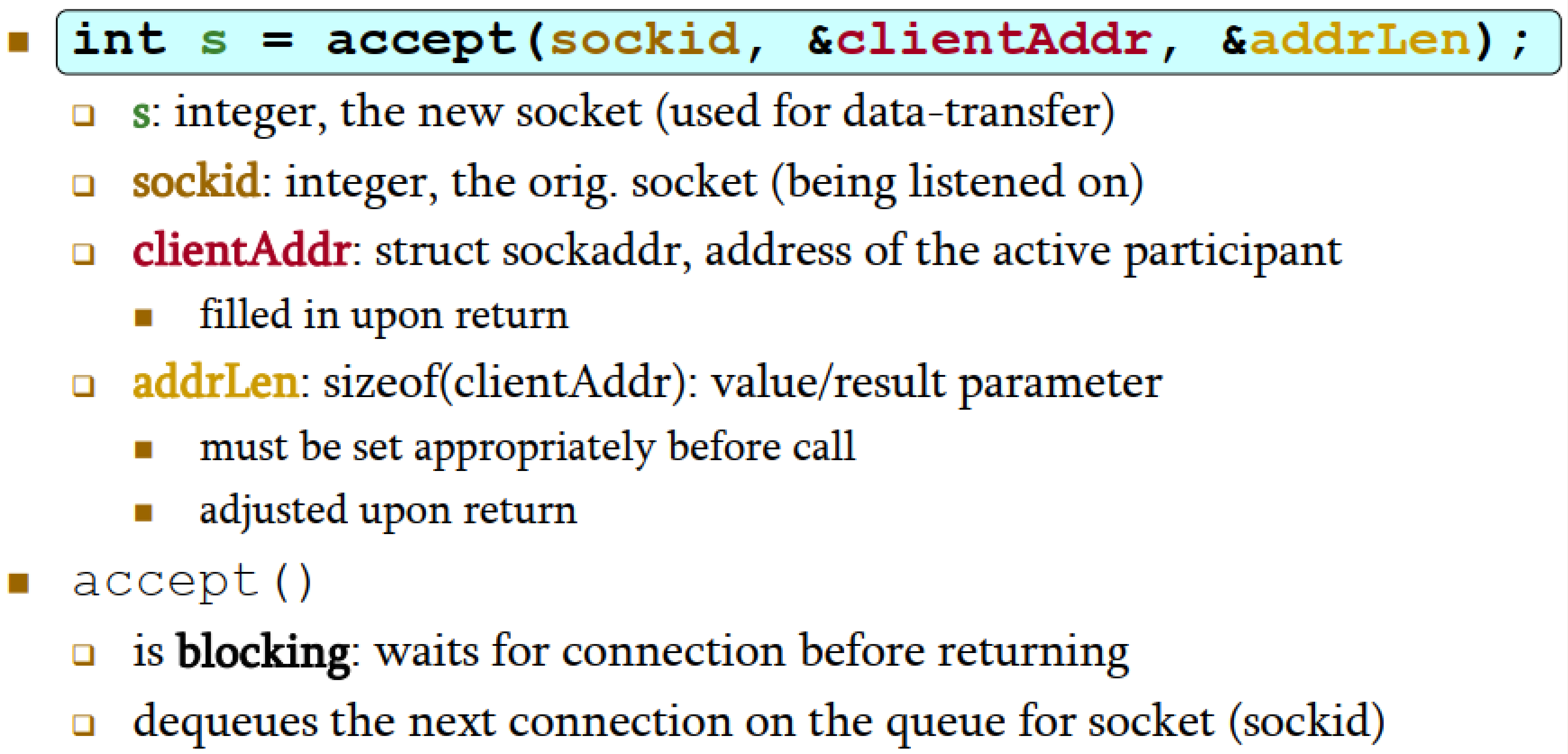
- The client establishes a connection with the server by calling connect()



- `int status = connect(sockid, &foreignAddr, addrlen);`
  - sockid: integer, socket to be used in connection
  - foreignAddr: struct sockaddr: address of the passive participant
  - addrlen: integer, sizeof(name)
  - status: 0 if successful connect, -1 otherwise
- `connect()` is **blocking**

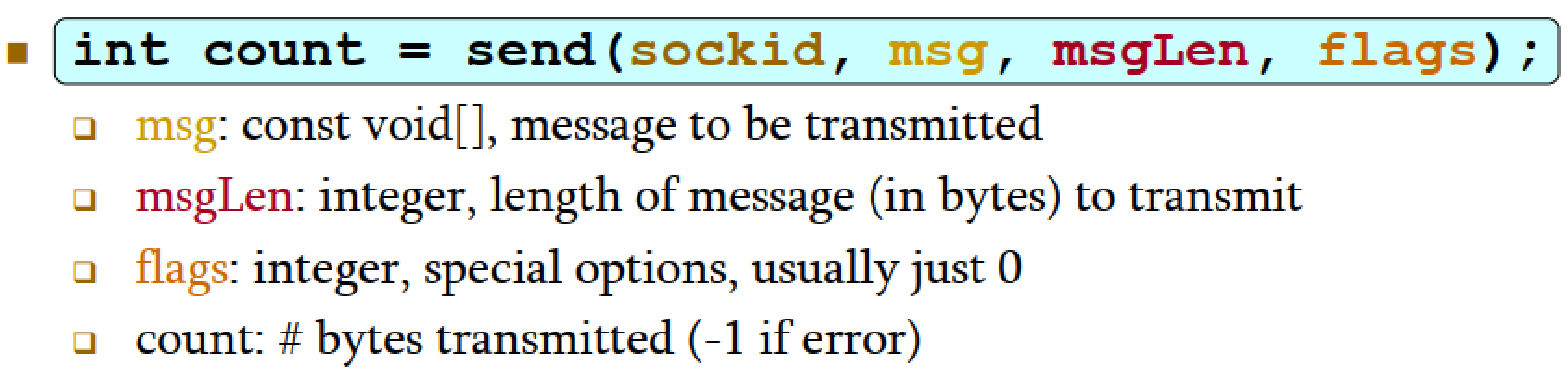
Incoming Connection: accept()

- The server gets a socket for an incoming client connection by calling accept()

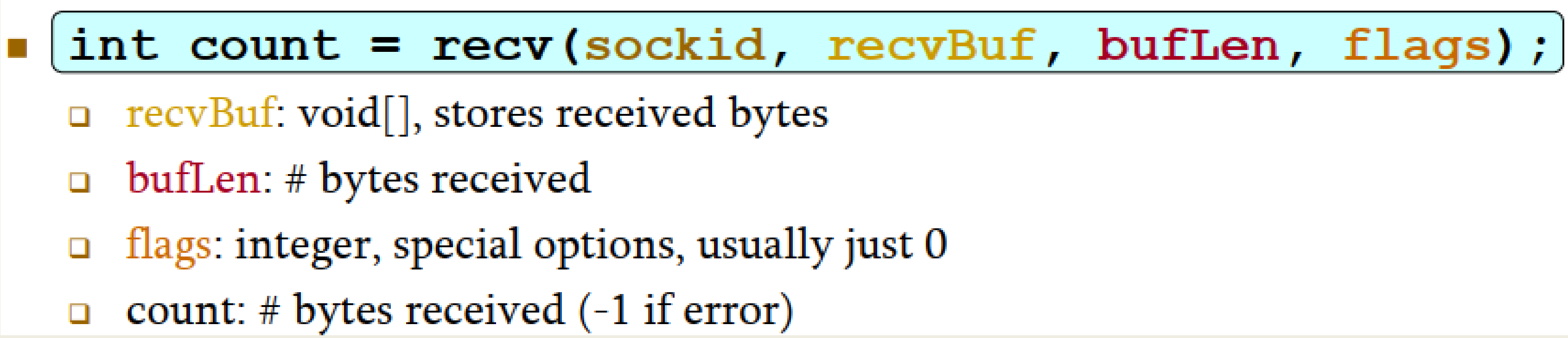


-
  ```
  int s = accept(sockid, &clientAddr, &addrLen);
  ```
  - s: integer, the new socket (used for data-transfer)
  - sockid: integer, the orig. socket (being listened on)
  - clientAddr: struct sockaddr, address of the active participant
    - filled in upon return
  - addrLen: sizeof(clientAddr): value/result parameter
    - must be set appropriately before call
    - adjusted upon return
- `accept()`
  - is **blocking**: waits for connection before returning
  - dequeues the next connection on the queue for socket (sockid)

Exchanging data with stream socket



- `int count = send(sockid, msg, msgLen, flags);`
  - msg: const void[], message to be transmitted
  - msgLen: integer, length of message (in bytes) to transmit
  - flags: integer, special options, usually just 0
  - count: # bytes transmitted (-1 if error)



- `int count = recv(sockid, recvBuf, bufLen, flags);`
  - recvBuf: void[], stores received bytes
  - bufLen: # bytes received
  - flags: integer, special options, usually just 0
  - count: # bytes received (-1 if error)

- Calls are blocking
  - returns only after data is sent / received

Reference

- Introduction to Sockets Programming in C using TCP/IP
  - Panagiota Fatourou