ICS 2305 : SYSTEMS PROGRAMMING

NETWORKING

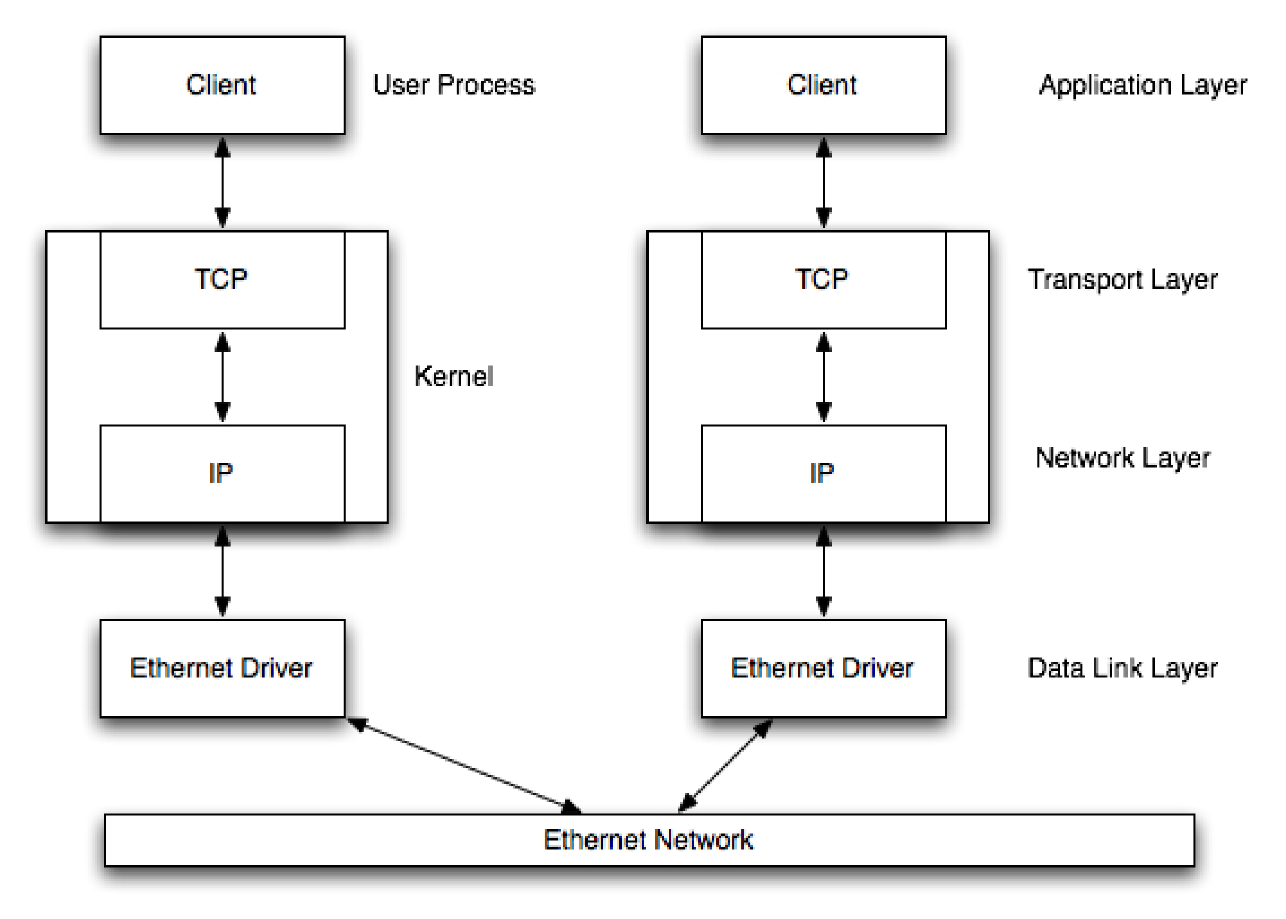
Karanja Mwangi

Lesson objective

- **At the end of this class you will**
  - Refresh on Networking concepts and how they relate to systems programming
  - Able to build your client and server (TCP) applications in C
  - Use C programs to interact with Ports

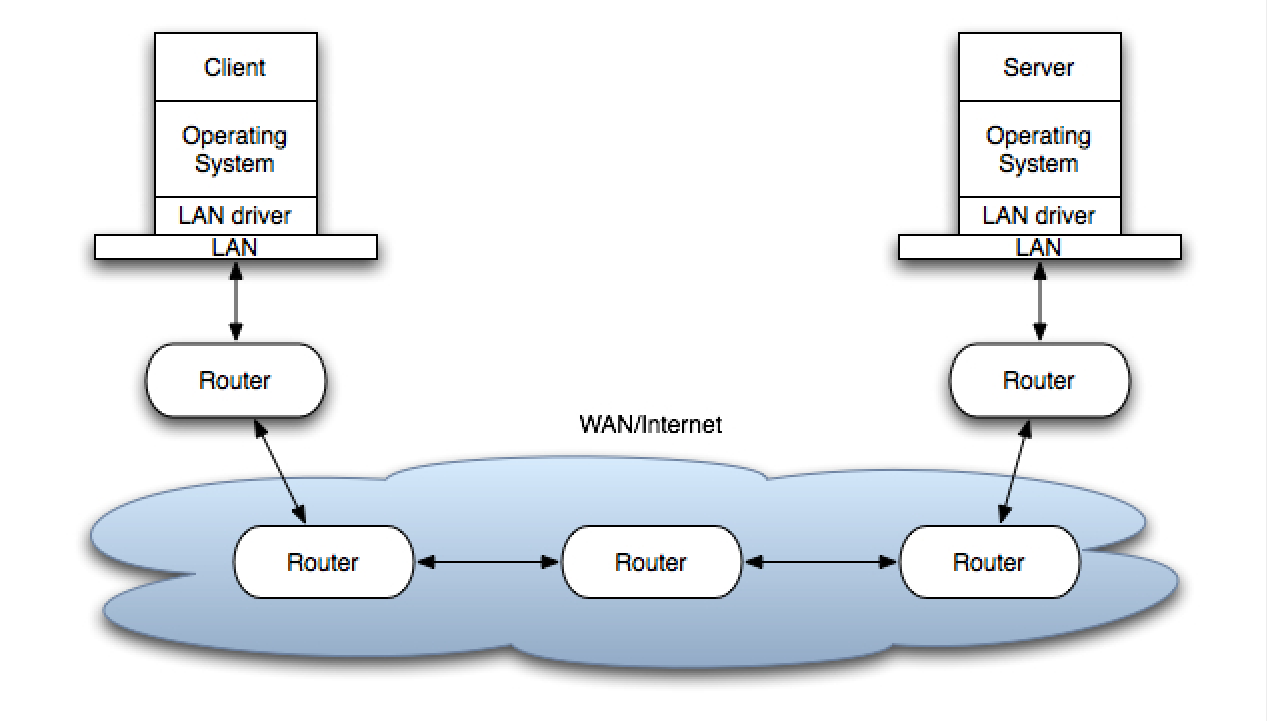
Same LAN

client and the server on the same local network

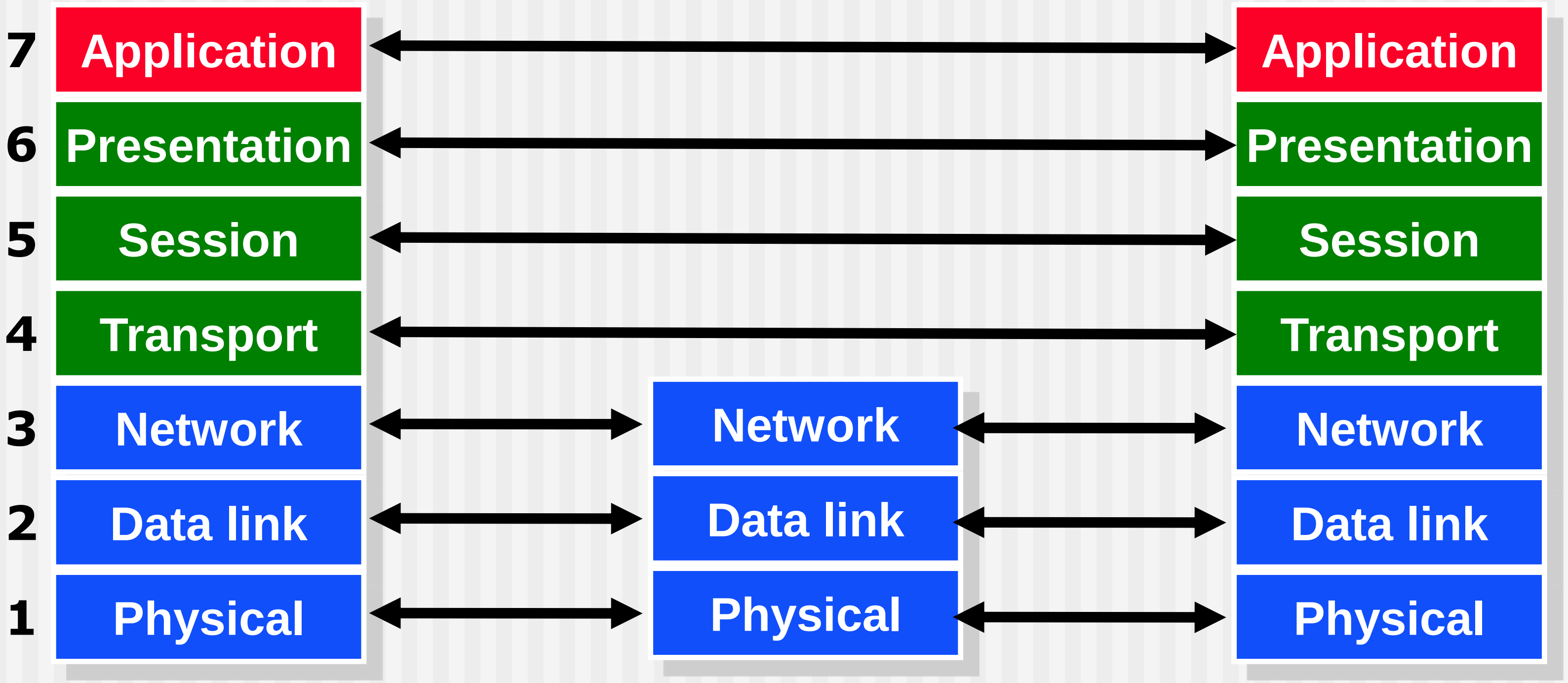


TCP/IP

The client and the server may be in different LANs, with both LANs connected to a Wide Area Network (WAN) by means of *routers*.



Network Layering



Client Server Communication TCP and UDP

- The main transport protocols TCP and UDP enables communication between network applications
- UDP is a connectionless protocol (RFC 768)
  - built on top of IPv4 and IPv6.
  - Determines the destination address and port and send your data packet!
  - Packets or Datagrams may be dropped if the network is congested. Packets may be duplicated or arrive out of order.--- No guarantee on good delivery

Client Server Communication-TCP

TCP is a connection-based protocol that is built on top of IPv4 and IPv6 ("TCP/IP" or "TCP over IP"). source IP address and source port number.

- TCP -Manages the packets, re-arranging out-of-order packets and changing the rate at which packets are sent, removes the duplicate etc
- TCP manages packets through 3 handshake SYN, SYN-ACK, and ACK
- For more on handshaking read https://www.inetdaemon.com/tutorials/internet/tcp/3-way_handshake.shtml

  - RFC 793, RFC 1323, RFC 2581 and RFC 3390.

User Datagram Protocol(UDP): An Analogy

UDP

- Single socket to receive messages
- No guarantee of delivery
- Not necessarily in-order delivery
- Datagram – independent packets
- Must address each packet

Postal Mail

- Single mailbox to receive letters
- Not very reliable
- Not necessarily in-order delivery
- Letters sent independently
- Must address each reply

Example UDP applications

Multimedia, voice over IP

Transmission Control Protocol (TCP): An Analogy

TCP

- Reliable – guarantee delivery
- Byte stream – in-order delivery
- Connection-oriented – single socket per connection
- Setup connection followed by data transfer

Telephone Call

- Guaranteed delivery
- In-order delivery
- Connection-oriented
- Setup connection followed by conversation

Example TCP applications

Web, Email, Telnet

PORTS (IPv4 vs. IPv6)

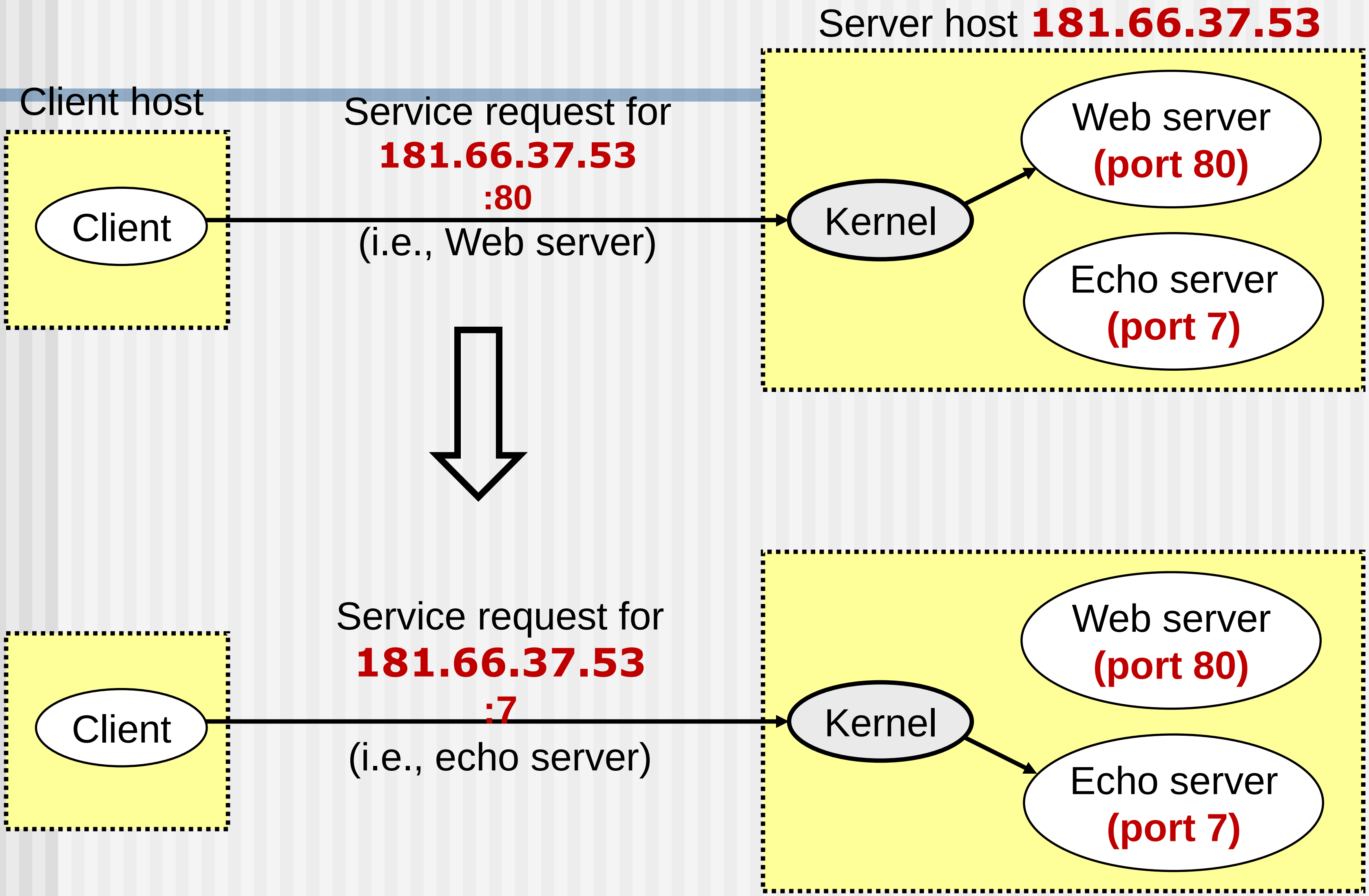
- Port - A 16-bit number that identifies the application process that receives an incoming message.
- Local hosts for IPv4 address is 127.0.0.1
- Local hosts for IPv6 address is IPv6 is 0:0:0:0:0:0:0:1 shortened form is ::1
- Both are stored in an *IP address struct* of appropriate type(library functions in C usually hide them)
  - in_addr for IPv4
  - in6_addr for IPv6
- Port numbers divided into three categories
  - Well Known Ports 0-1023
  - Registered Ports 1024-49151 by the IANA (Internet Assigned Numbers Authority), and represent *second tier* common ports (socks (1080), WINS (1512), kermit (1649), https (443))
  - Dynamic/Private Ports 49152-65535 *ephemeral* ports, available for temporary client usage

Well Known Ports- Reserved ports

- Reserved ports or well-known ports (0 to 1023)
  - Standard ports for well-known applications.
  - See /etc/services file on any UNIX machine for listing of services on reserved ports.

1	TCP Port Service Multiplexer
7	Echo Server
20	File Transfer Protocol (FTP) Data
21	FTP Control
23	Telnet
25	Simple Mail Transfer (SMT)
43	Who Is
69	Trivial File Transfer Protocol (TFTP)
80	HTTP

Using Ports to Identify Services



The Socket----Socket as a File

- a socket is like a file: you can read/write to/from the network just like you would a file
- For connection-oriented communication (e.g. TCP)
  - servers (passive open) do listen and accept operations
  - clients (active open) do connect operations
  - both sides can then do read and/or write (or send and recv)
  - then each side must close etc
- Connectionless (e.g. UDP): uses sendto and recvfrom

Sockets And Socket Libraries

- **In Unix, socket procedures** (e.g. listen, connect, etc.) **are *system calls***
  - part of the operating system
  - when you call the function, control moves to the operating system, and you are using “system” CPU time

Sockets And Socket Libraries

- **On some operating systems, socket procedures are *not* part of the OS e.g in Windows**
  - instead, they are implemented as a library, linked into the application object code (e.g. a DLL under Windows)
  - Typically, this DLL makes calls to similar procedures that are part of the native operating system.

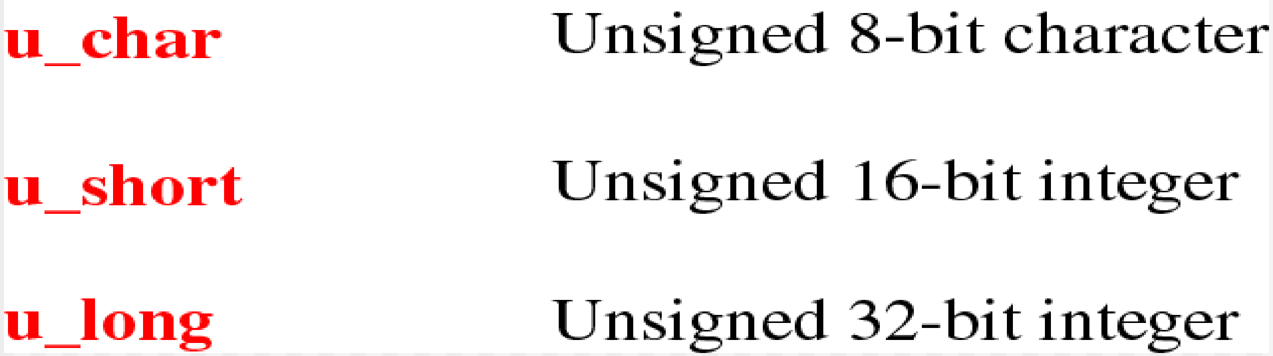
The Most Popular Socket Interface

- The ***Berkeley Sockets API***
  - Originally developed as part of BSD Unix
  - BSD = Berkeley Software Distribution
    - API=Application Program Interface
  - Now the most popular API for C/C++ programmers writing applications over TCP/IP
    - Also emulated in other languages: Perl, Tcl/Tk, Python etc.
    - Also emulated on other operating systems: Windows, etc.

Sockets and Data types

- **Data types**

int8_t	signed 8-bit integer
int16_t	signed 16-bit integer
int32_t	signed 32-bit integer
uint8_t	unsigned 8-bit integer
uint16_t	unsigned 16-bit integer
uint32_t	unsigned 32-bit integer

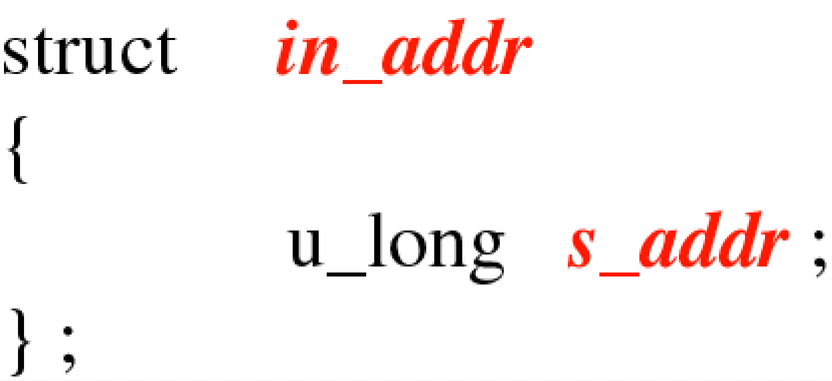


u_char	Unsigned 8-bit character
u_short	Unsigned 16-bit integer
u_long	Unsigned 32-bit integer

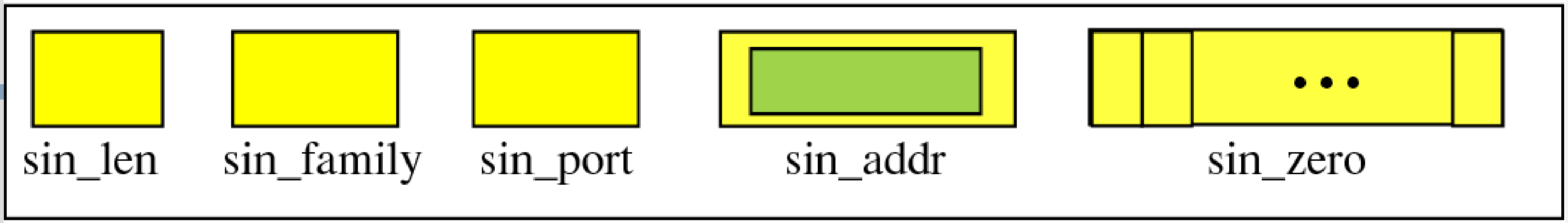
More on Data types

- Internet Address Structure

```
struct in_addr
{
	in_addr_t	s_addr;
};
```



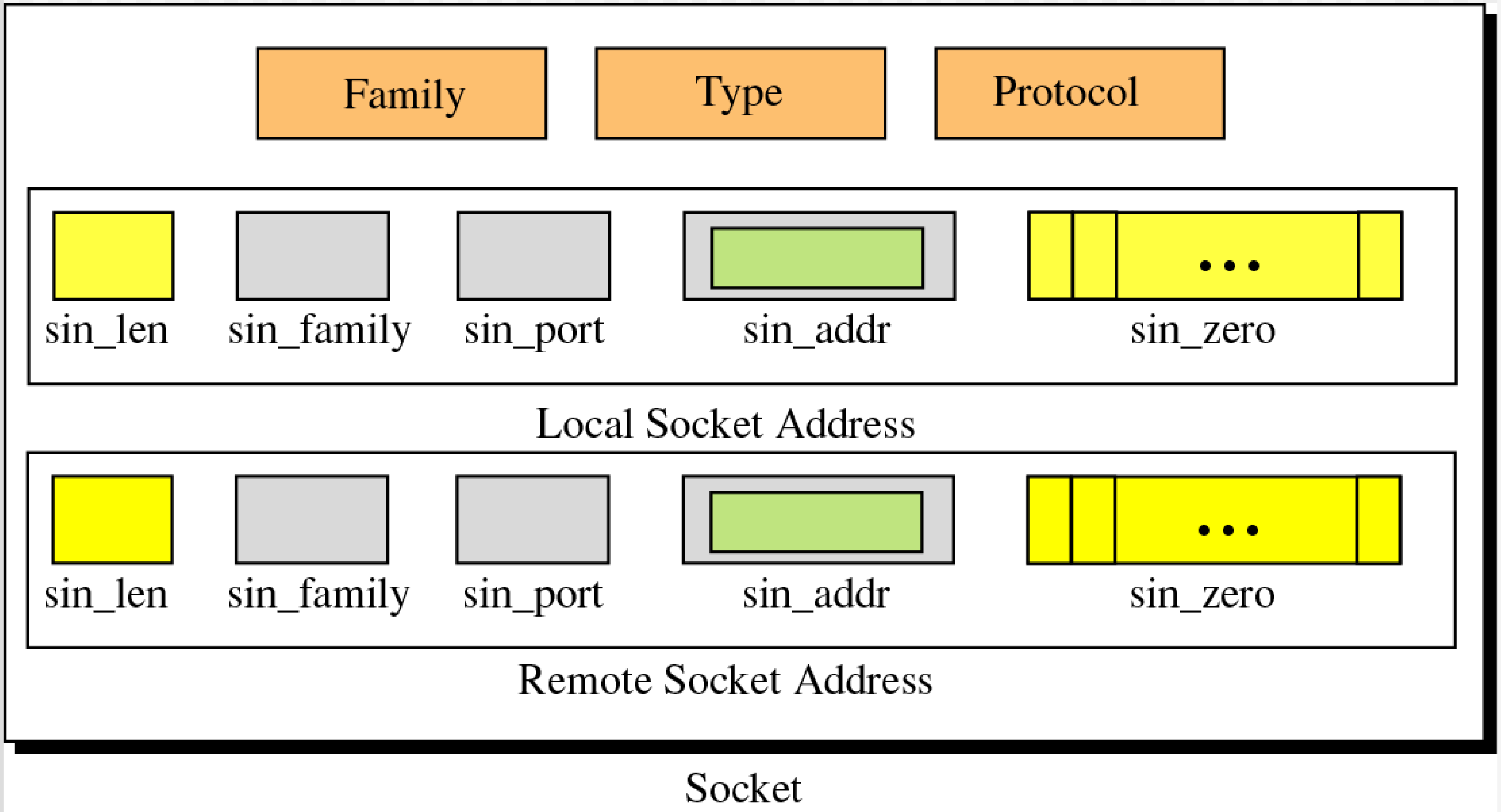
```
struct    in_addr
{
          u_long  s_addr ;
} ;
```



sockaddr_in

```
struct    sockaddr_in
{
          u_char                  sin_len ;
          u_short                 sin_family ;
          u_short                 sin_port ;
          struct  in_addr         sin_addr ;
          char                    sin_zero [8] ;
} ;
```

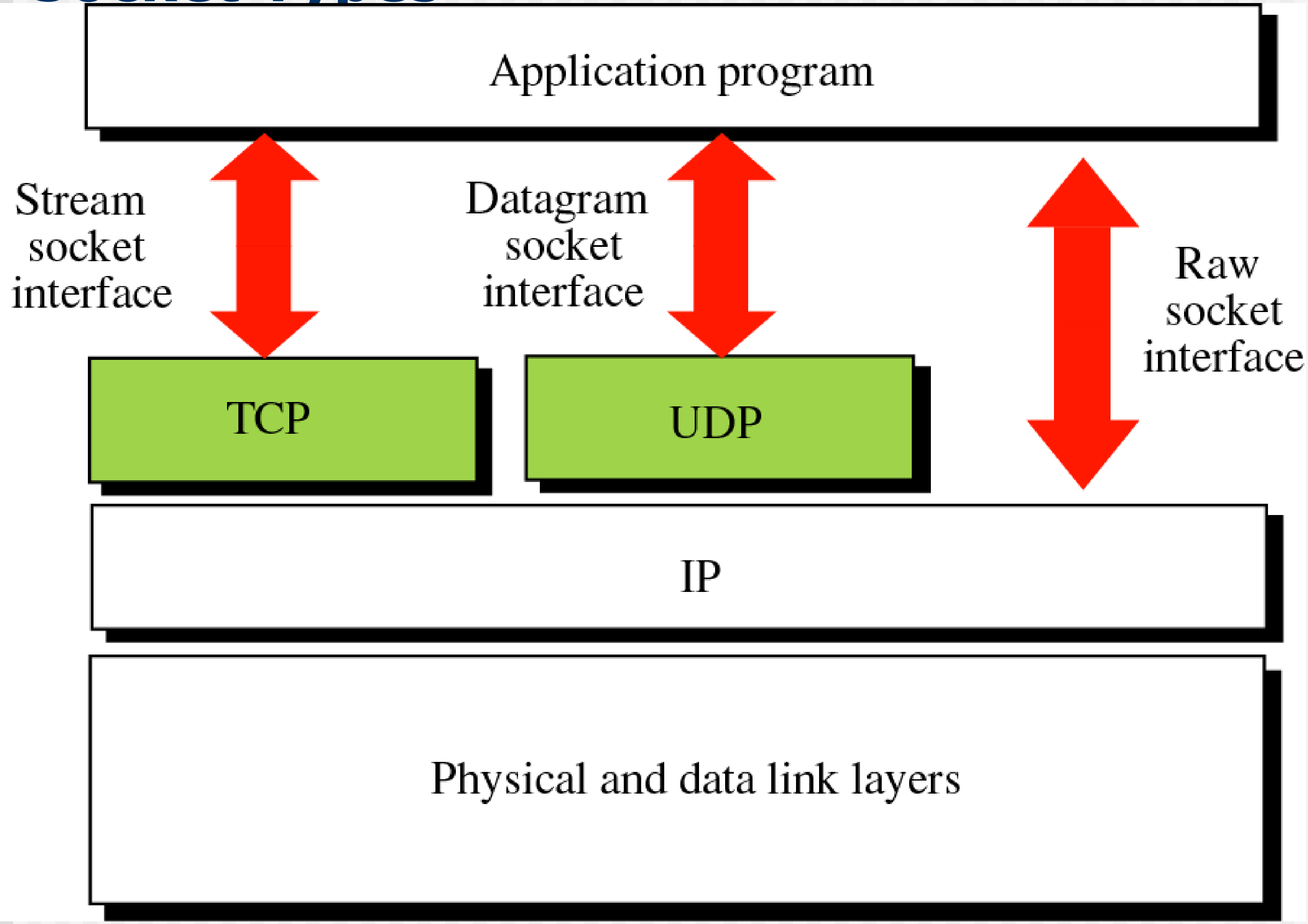
Socket Structure



Sockets

- Sockets provide a standard interface between network and application –they are Independent of network type:
- Commonly used with TCP/IP and UDP/IP,
- Two types of socket:
  - Stream – provides a virtual circuit service
  - Datagram – delivers individual packets

Socket Types

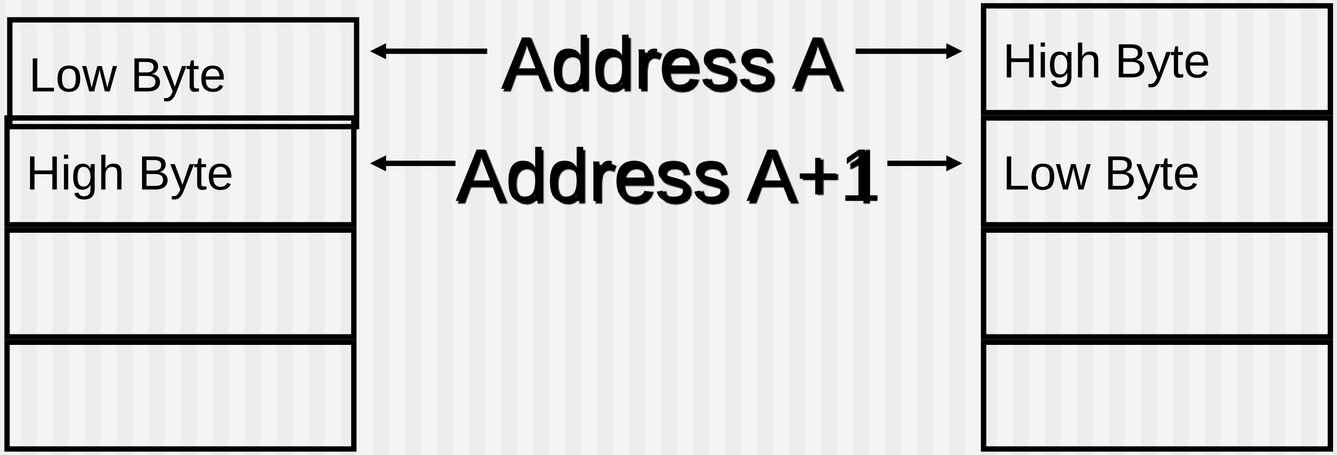


Byte Ordering

- two ways to store two bytes in memory: with the lower-order byte at the starting address (*little-endian* byte order) or with the high-order byte at the starting address (*big-endian* byte order). They are called *host byte order*.
  - an Intel processor stores the 32-bit integer as four consecutives bytes in memory in the order 1-2-3-4, where 1 is the most significant byte. IBM PowerPC processors would store the integer in the byte order 4-3-2-1.

Big-Endian (RISC-Sparc)

Little-Endian (Intel)



Low Byte

High Byte

Address A

Address A+1

High Byte

Low Byte

Byte Order and Networking

- Suppose a Big Endian machine sends a 16 bit integer with the value 2:

0000000000000010

- A Little Endian machine will understand the number as 512:

0000001000000000

- How do two machines with different byte-orders communicate?
  - Using network byte-order
  - Network byte-order = big-endian order

network byte-order conversion

- The following functions are used for conversion to the network order
- **The htons(), htonl(), ntohs(), and ntohl() Functions**

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

uint16_t htons(uint16_t host16bitvalue);

uint32_t htonl(uint32_t host32bitvalue);

uint16_t ntohs(uint16_t net16bitvalue);

uint32_t ntohl(uint32_t net32bitvalue);
```

  - first two return the value in network byte order (16 and 32 bit, respectively). The latter return the value in host byte order (16 and 32 bit, respectively).

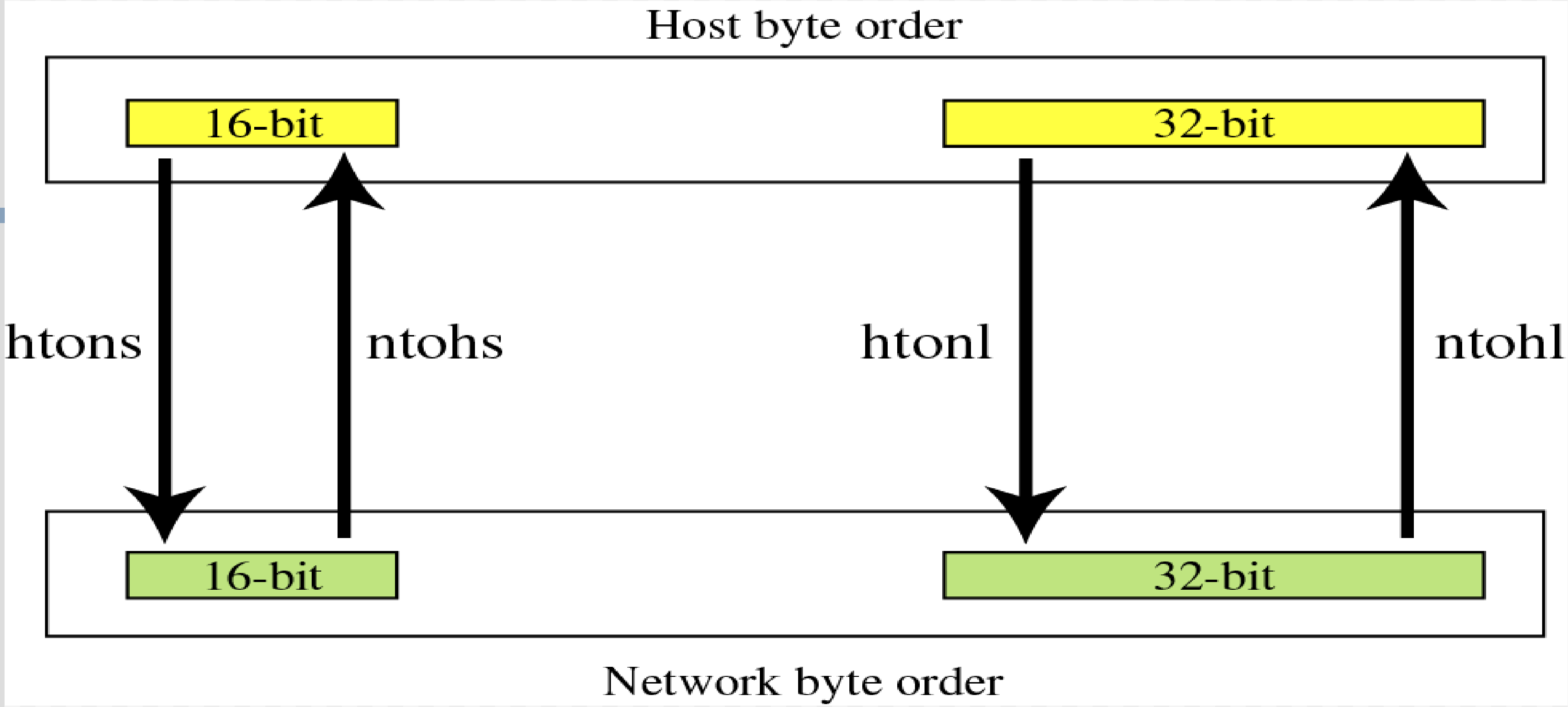
Network Byte Order

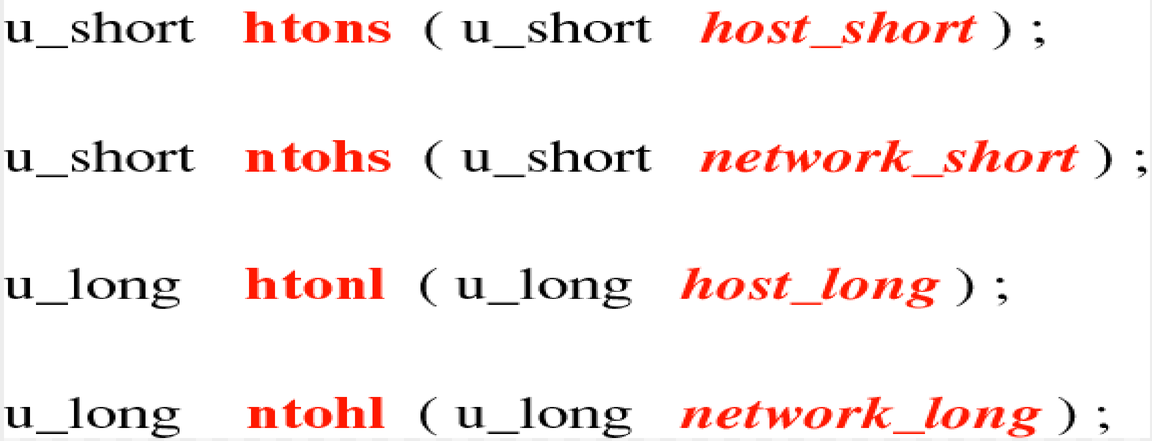
- Conversion of application-level data is left up to the presentation layer.
- Lower level layers communicate using a fixed byte order called *network byte order* for all control data.
- TCP/IP mandates that *big-endian* byte ordering be used for transmitting protocol information
- All values stored in a `sockaddr_in` must be in network byte order.
  - `sin_port` a TCP/IP port number.
  - `sin_addr` an IP address.

Network Byte Order Functions

- Several functions are provided to allow conversion between host and network byte ordering,
- Conversion macros `(<netinet/in.h>)`
  - to translate 32-bit numbers (i.e. IP addresses):
    - unsigned long htonl(unsigned long hostlong);
    - unsigned long ntohl(unsigned long netlong);
  - to translate 16-bit numbers (i.e. Port numbers):
    - unsigned short htons(unsigned short hostshort);
    - unsigned short ntohs(unsigned short netshort);

Byte-Order Transformation

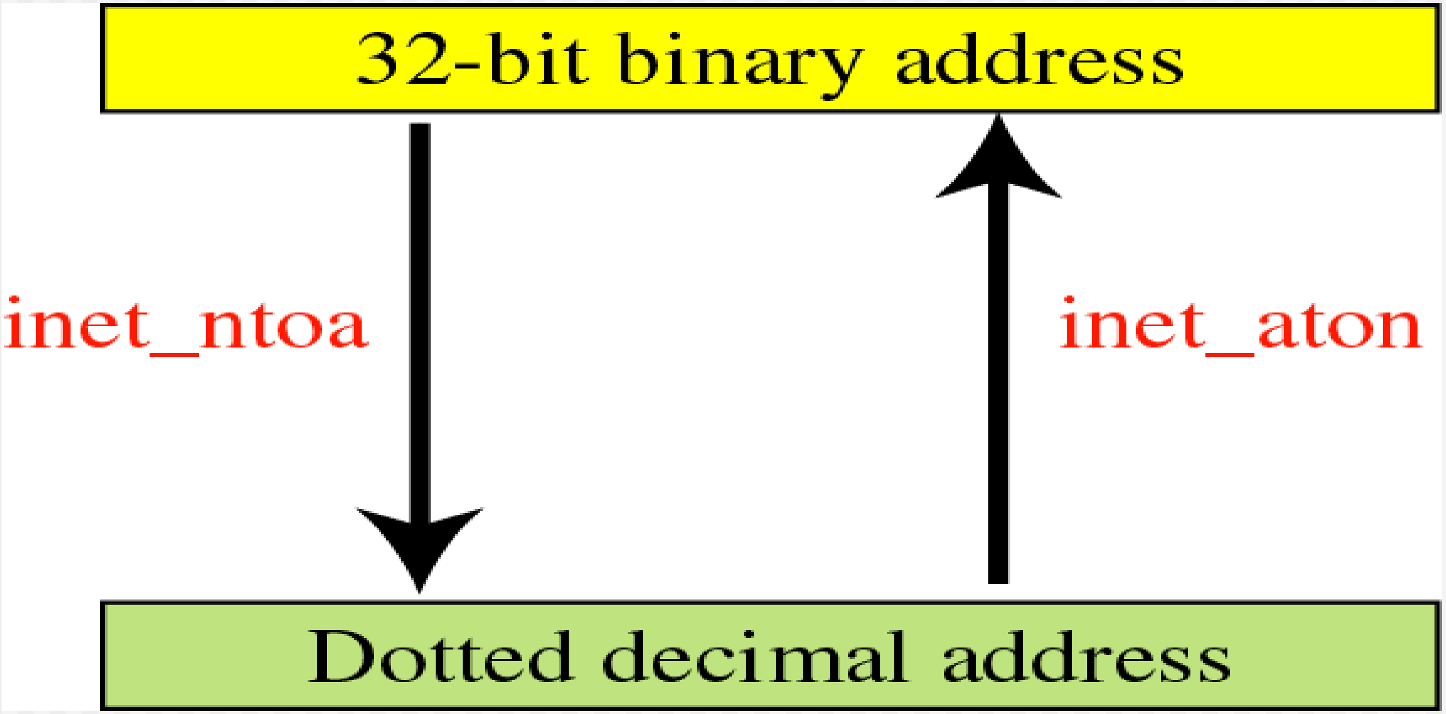




```
u_short   htons  ( u_short   host_short ) ;

u_short   ntohs  ( u_short   network_short ) ;

u_long    htonl  ( u_long   host_long ) ;

u_long    ntohl  ( u_long   network_long ) ;
```

Address Transformation

```
int           inet_aton  ( const char   *strptr , struct in_addr  *addrptr ) ;

char      *inet_ntoa  (struct in_addr   inaddr ) ;
```



32-bit binary address

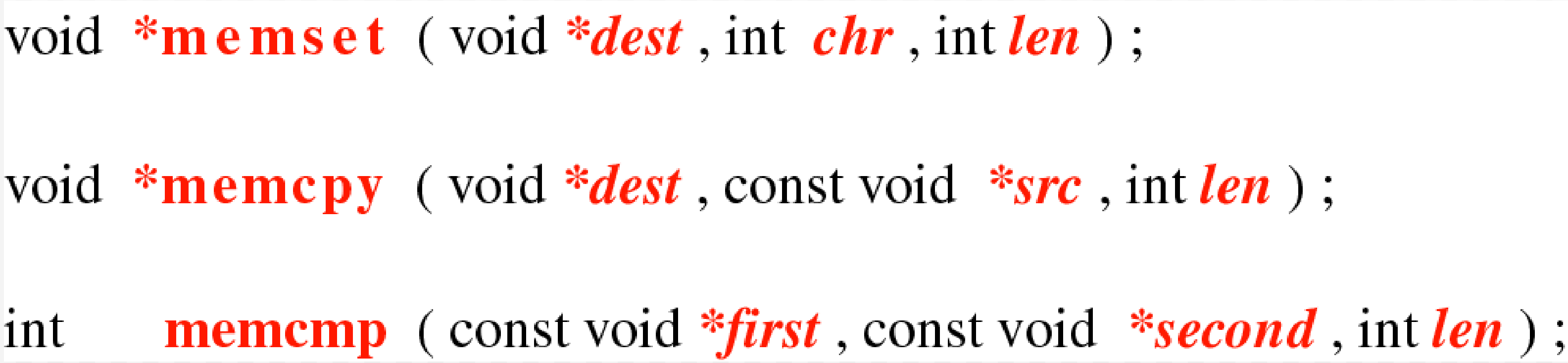
inet_ntoa

inet_aton

Dotted decimal address

Byte-Manipulation Functions

- In network programming, we often need to initialize a field, copy the contents of one field to another, or compare the contents of two fields.
    - Cannot use string functions (strcpy, strcmp, …) which assume null character termination.



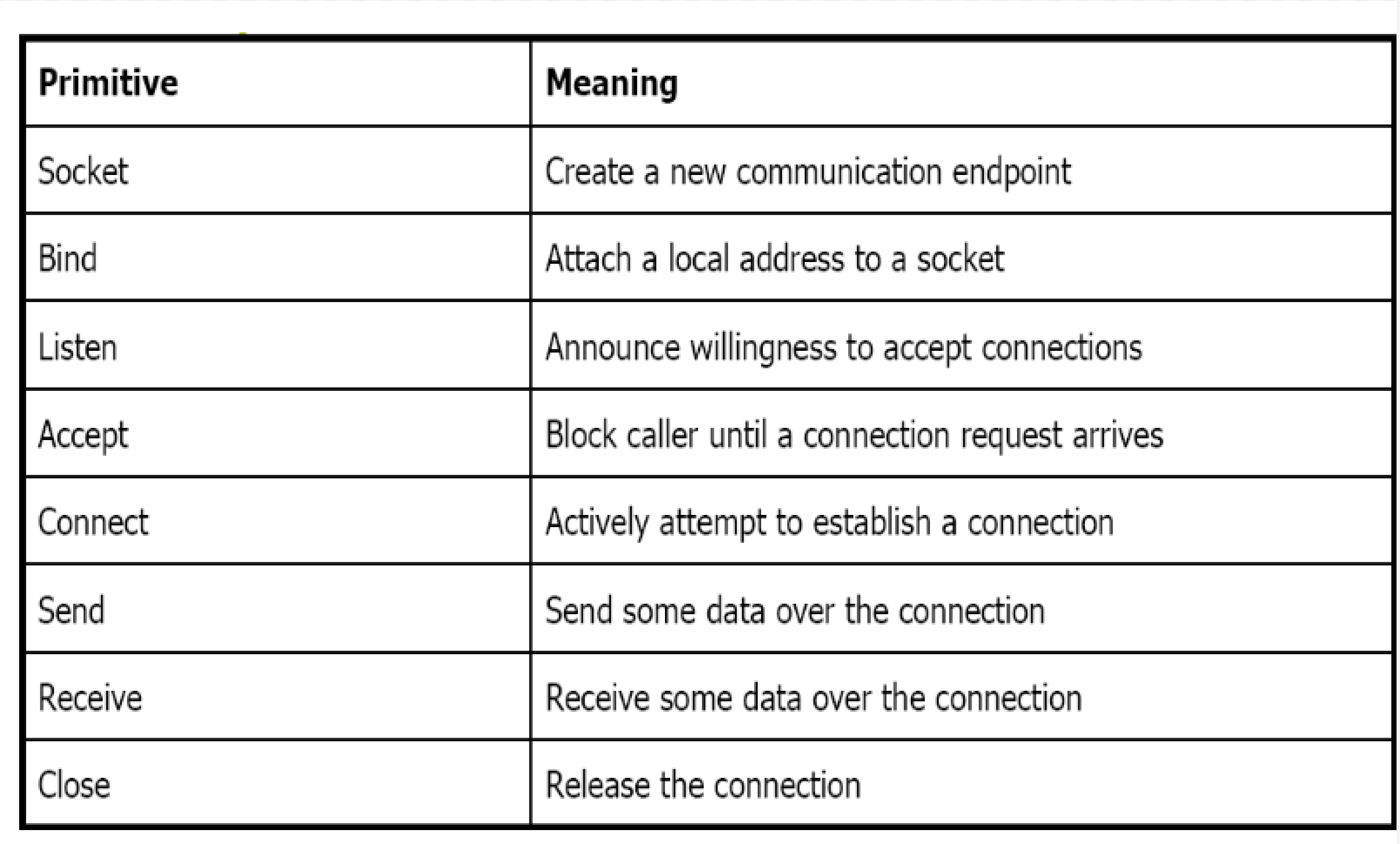
```
void  *memset  ( void *dest , int  chr , int len ) ;

void  *memcpy  ( void *dest , const void  *src , int len ) ;

int      memcmp  ( const void *first , const void  *second , int len ) ;
```

Creating a Socket: The socket() Function

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

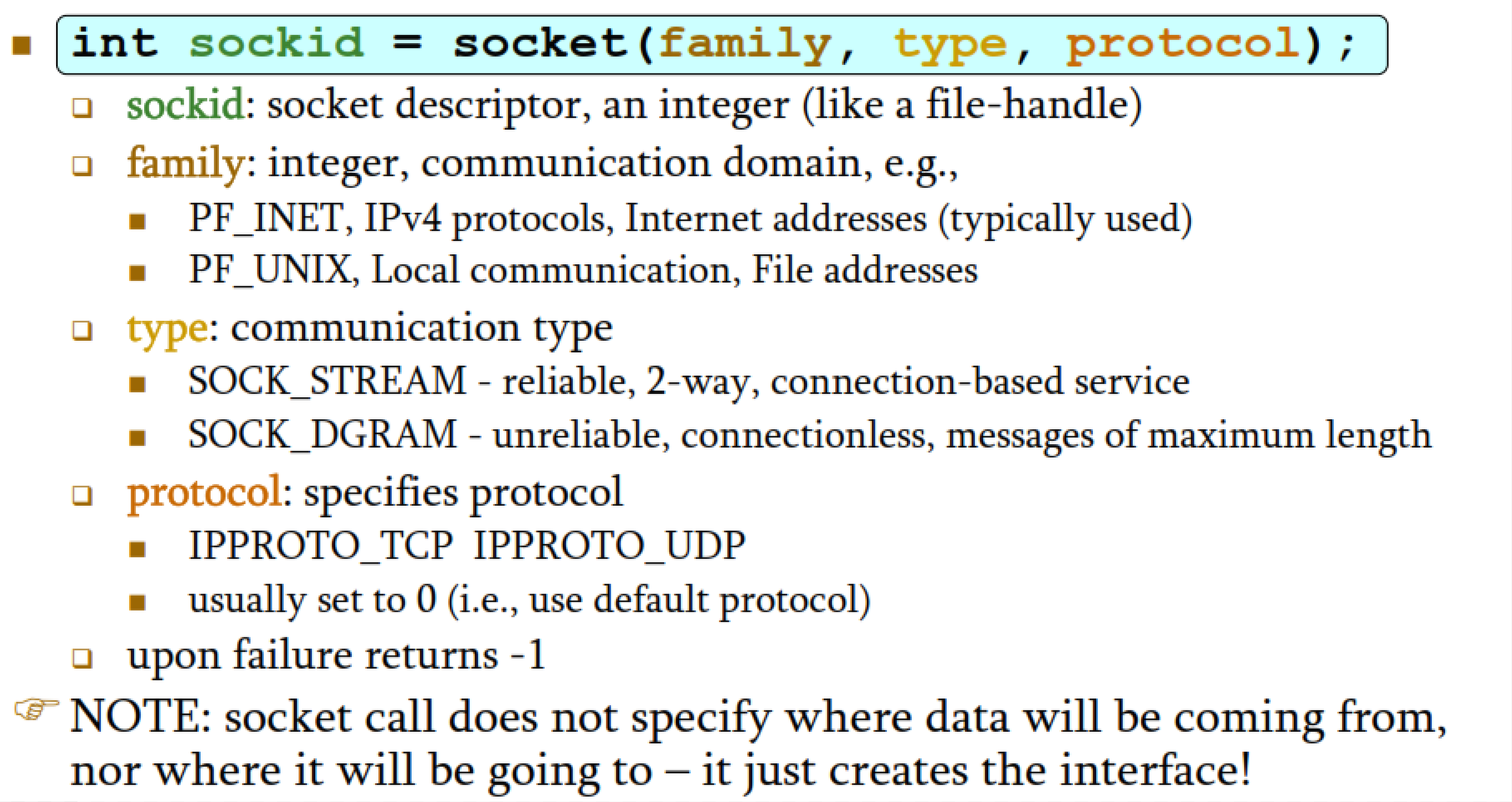
- domain is one of the *Protocol Families* (AF_INET, AF_UNIX, etc.)
- type defines the communication protocol semantics, usually defines either:
  - SOCK_STREAM: connection-oriented stream (TCP)
  - SOCK_DGRAM: connectionless, unreliable (UDP)
- protocol specifies a particular protocol, just set this to 0 to accept the default
- Example **int sockfd = socket (AF_INET, SOCK_STREAM, 0);**

Socket Primitives ---in the library



Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

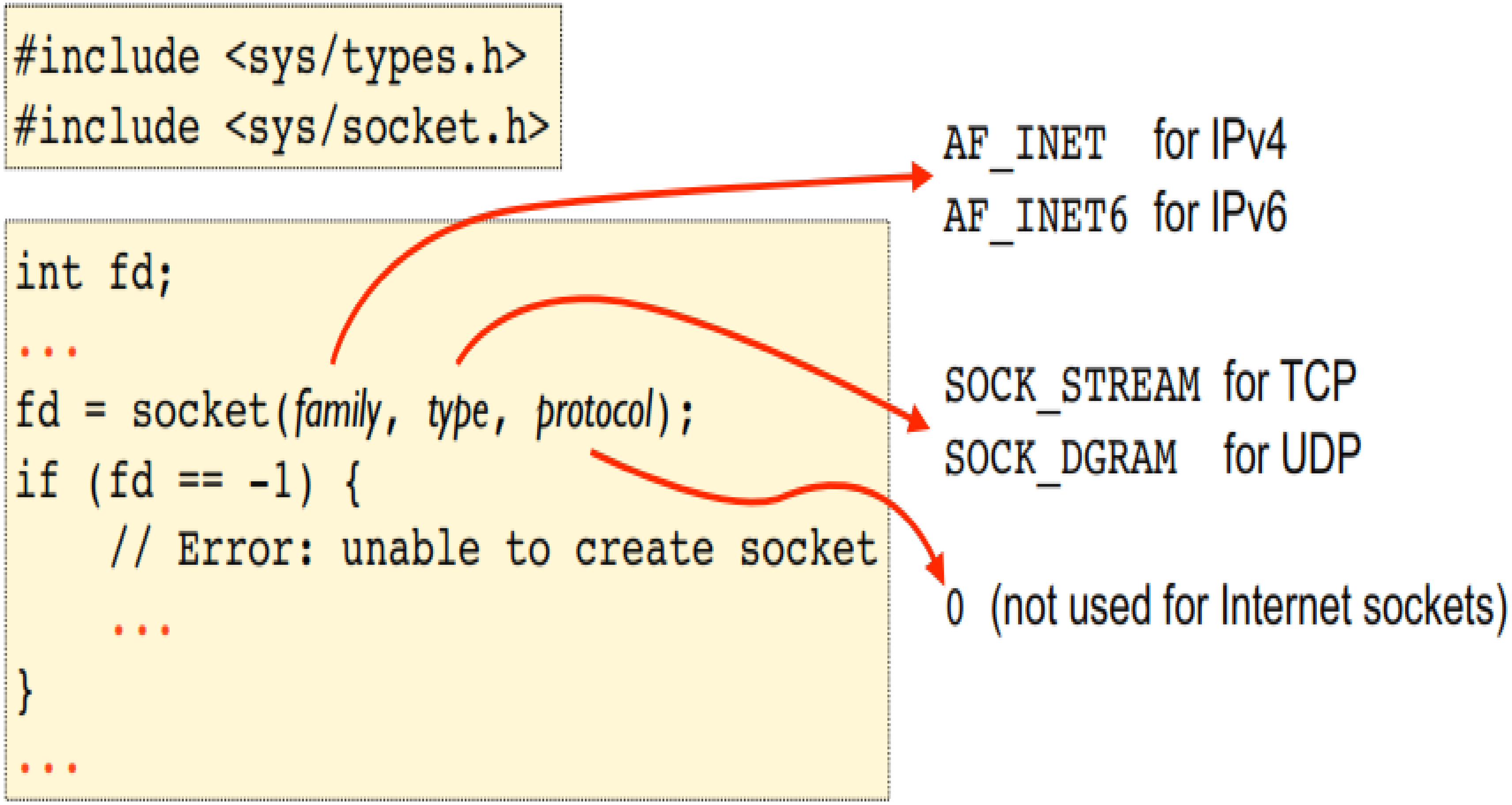
Creating a Socket-2:The socket() Function



- `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!

Programming Sockets



```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int fd;
...
fd = socket(family, type, protocol);
if (fd == -1) {
    // Error: unable to create socket
    ...
}
...
```

AF_INET for IPv4

AF_INET6 for IPv6

SOCK_STREAM for TCP

SOCK_DGRAM for UDP

0 (not used for Internet sockets)

Specifying Addresses

- Socket API defines a generic data type for addresses:

```
struct sockaddr {
   unsigned short sa_family;  /* Address family (e.g. AF_INET) */
   char sa_data[14];          /* Family-specific address information */
}
```

- Particular form of the sockaddr used for TCP/IP addresses:

```
struct in_addr {
   unsigned long s_addr;         /* Internet address (32 bits) */
}
struct sockaddr_in {
   unsigned short sin_family;    /* Internet protocol (AF_INET) */
   unsigned short sin_port;      /* Address port (16 bits) */
   struct in_addr sin_addr;    /* Internet address (32 bits) */
   char sin_zero[8];             /* Not used */
}
```

☞ **Important**: sockaddr_in can be casted to a sockaddr

The connect() Function

- The connect() function is used by a TCP client to establish a connection with a TCP server

```
#include <sys/socket.h>

int connect (int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

- where sockfd is the socket descriptor returned by the socket function.
- The function **returns 0** if the it succeeds in establishing a connection (i.e., successful TCP three-way handshake**, -1 otherwise**.

The bind() Function

- The bind() assigns a local protocol address to a socket.

```
          #include <sys/socket.h>
          int bind(int sockfd, const struct sockaddr *servaddr, socklen_t addrl
en);
```

where sockfd is the socket descriptor, servaddr is a pointer to a protocol-specific address and addrlen is the size of the address structure.

bind() returns 0 if it succeeds, -1 on error.

The bind() Function-2 example in IPV4

- use of generic socket address sockaddr requires that any calls to these functions must cast the pointer to the protocol-specific address structure

```
struct sockaddr_in serv; /* IPv4 socket address structure */

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

- **A process can bind a specific IP address to its socket: for a TCP client, this assigns the source IP address that will be used for IP datagrams sent on the sockets. For a TCP server, this restricts the socket to receive incoming client connections destined only to that IP address**.
- TCP client does not bind an IP address to its socket. The kernel chooses the source IP socket to be connected, based on the outgoing interface that is used. If a TCP server does not bind an IP address to its socket, the kernel uses the destination IP address of the incoming packets as the server’s source address.

The listen() Function

The listen() function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.

```
#define _OE_SOCKETS
 #include <sys/socket.h>
 int listen(int socket, int backlog);
```

Where

Backlog Defines the maximum length for the queue of pending connections.

The function listen() return 0 if it succeeds, -1 on error.

The accept() Function

- used by a server to accept a connection request from a client

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *cliaddr,
socklen_t *addrlen);
```

sockfd is a new file descriptor that is connected to the client that called the connect()

cliaddr and addrlen arguments are used to return the protocol address of the client

The send() Function

- Remember Socket as a file ? We can use *read* and *write* to communicate with a socket as long as it is connected.
- If we need to specify other options , other than read and write plainly , we use other functions. E.g here send() is similar to write() but allows to specify some options. buf and nbytes have the same meaning as they have with write. flags is used to specify how we want the data to be transmitted

```
#include <sys/socket.h>
ssize_t send(int sockfd, const void *buf, size_t nbytes, i
nt flags);
```

The function returns the number of bytes if it succeeds, -1 on error

The receive() Function

- The recv() function is similar to read(), but allows to specify some options

```
#include <sys/socket.h>
ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
```

The function returns the length of the message in bytes, 0 if no messages are available and peer had done an orderly shutdown, or -1 on error.

The close() Function

- close() function is used to close a socket and terminate a TCP socket. It returns 0 if it succeeds, -1 on error.

```
#include <unistd.h>

int close(int sockfd);
```

More on Networking and Sockets

- functions ***getaddrinfo()*** and ***getnameinfo()*** convert do main names, hostnames, and IP addresses between human-readable text representations and structured binary formats for the operating system's networking API.

Working Exercises

- To be shared in the Piazza

Sources

- https://beej.us/guide/bgnet/html//index.html
- https://academy.nordicsemi.com/courses/cellular-iot-fundamentals/lessons/lesson-3-cellular-fundamentals/topic/lesson-3-exercise-1/
- https://www.youtube.com/watch?v=WdE3PCHSBy8
- https://people.cs.rutgers.edu/~pxk/rutgers/notes/sockets/