CS354R

DR SARAH ABRAHAM

SOCKET PROGRAMMING

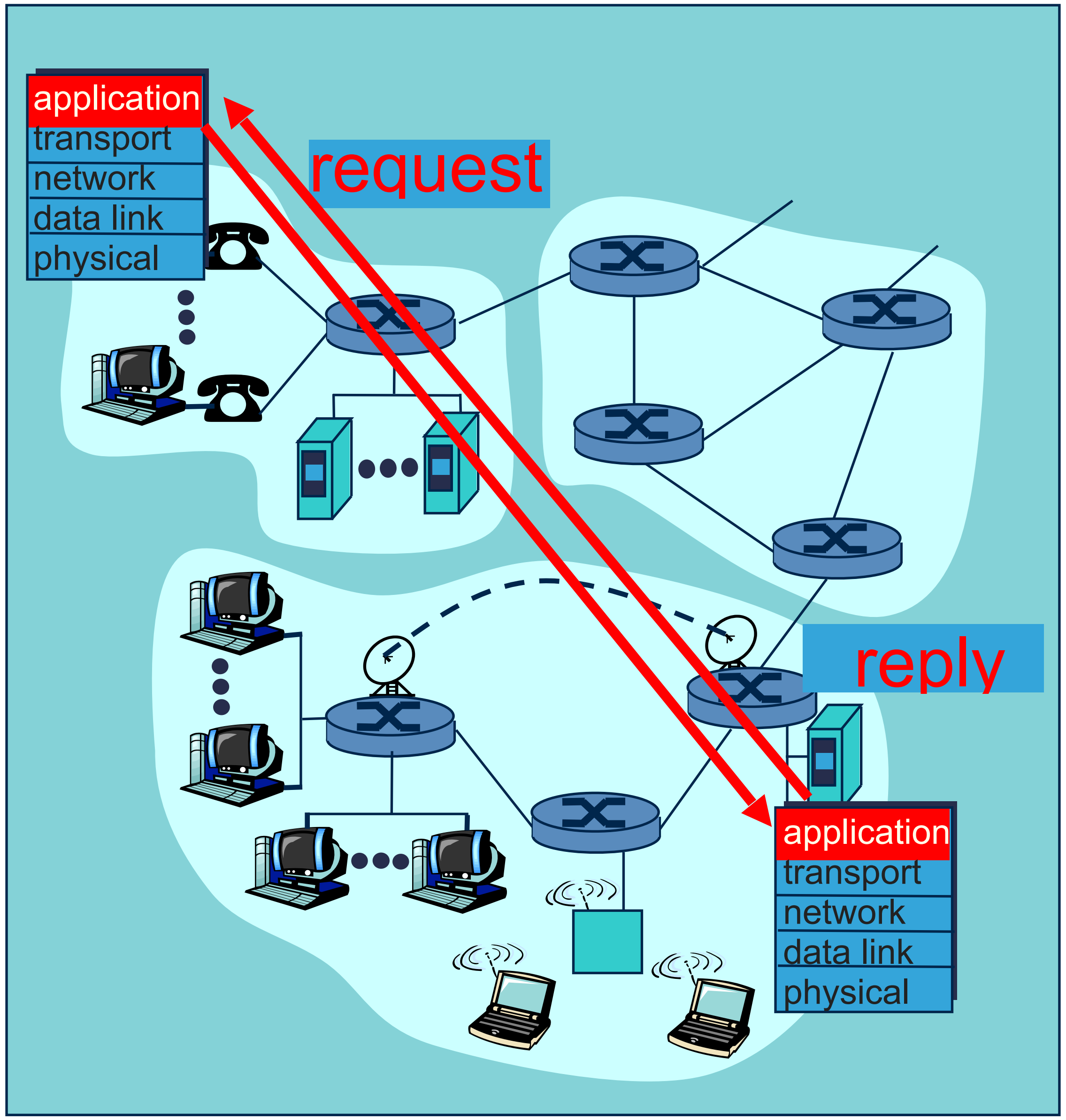
LECTURE OVERVIEW

- Application layer
  - Client-server
  - Application requirements
- Background
  - TCP vs. UDP
  - Byte ordering
- Socket I/O
  - TCP/UDP server and client
  - I/O multiplexing

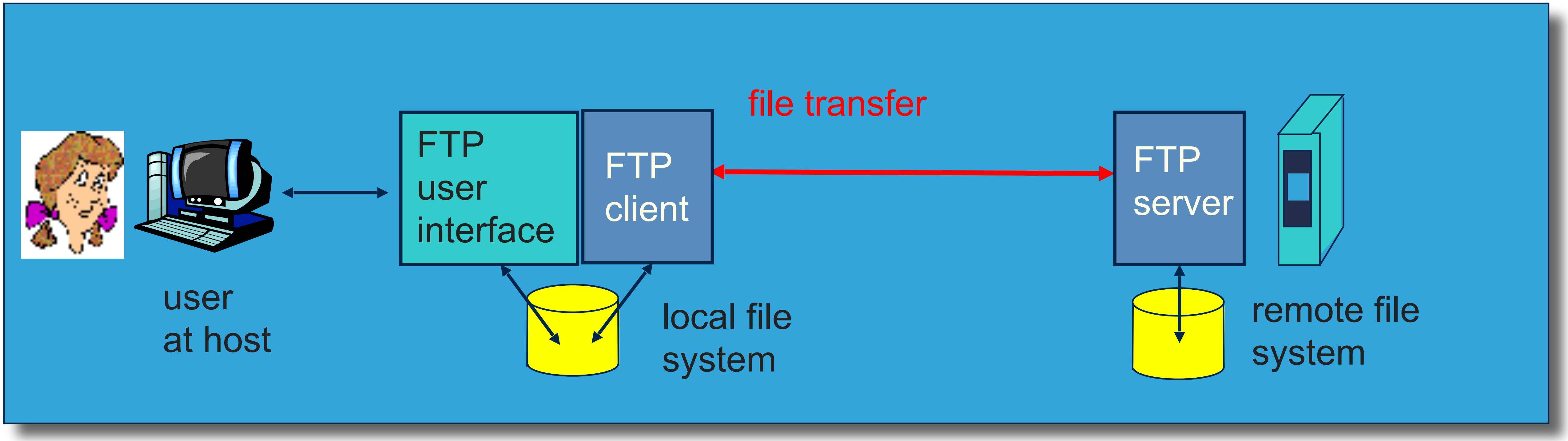
CLIENT-SERVER PARADIGM

Typical network app has two pieces: client and server

- Client
  - Initiates contact with server
  - Typically requests service from server
  - Client implemented in browser for web, mail reader for e-mail
- Server
  - Provides requested service to client
  - e.g. Sends web page, delivers e-mail



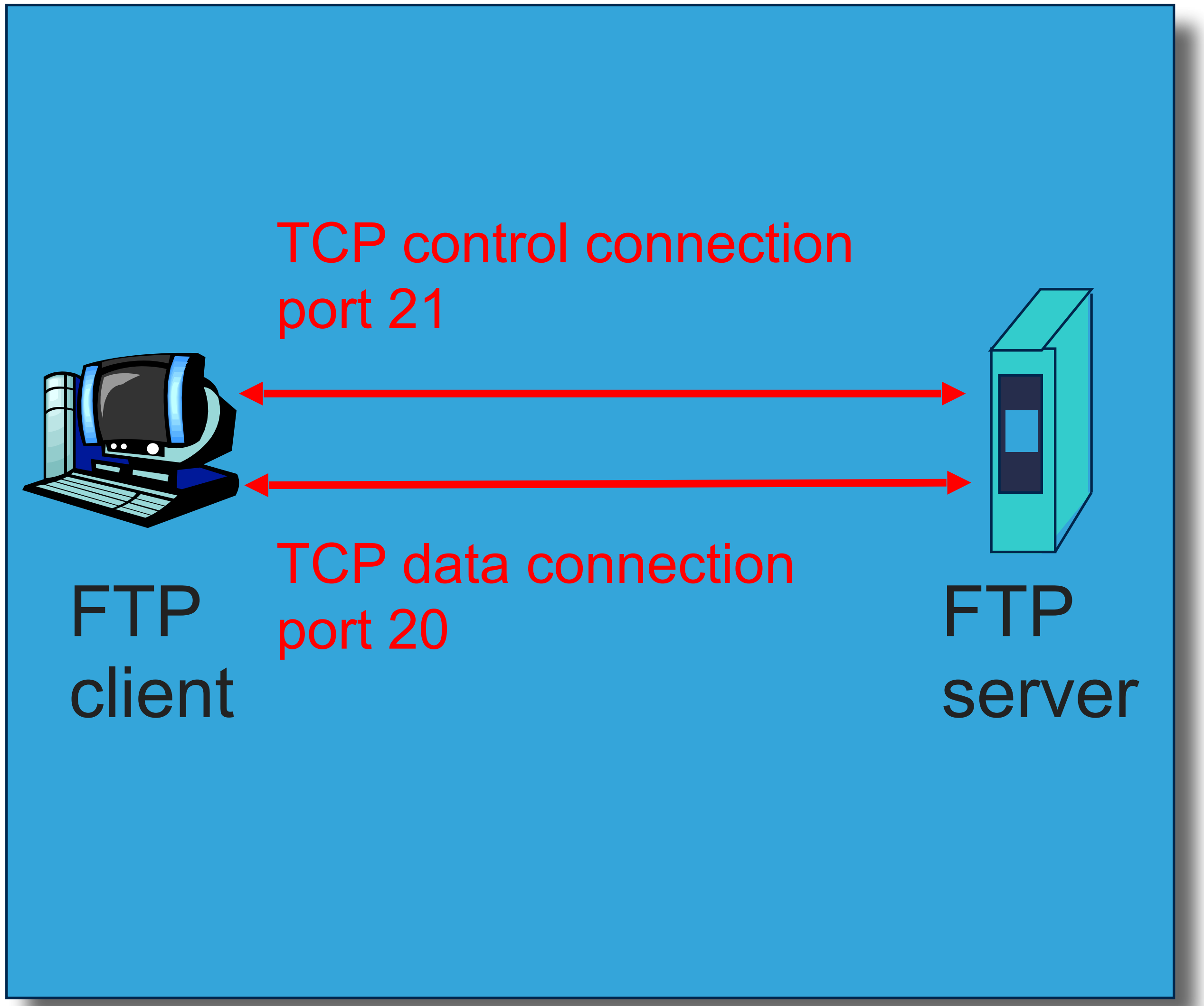
FTP: THE FILE TRANSFER PROTOCOL



- Transfer file to/from remote host
- Client/server model
  - Client: side that initiates transfer (either to/from remote)
  - Server: remote host
- ftp: RFC 959
- ftp server: port 21

SEPARATE CONTROL, DATA CONNECTIONS

- Ftp client contacts ftp server at port 21, specifying TCP as transport protocol
- Two parallel TCP connections opened:
  - Control: exchange commands, responses between client and server
  - Data: file data to/from server
  - Out-of-band protocol
- Ftp server maintains “state”: current directory, earlier authentication



FTP COMMANDS, RESPONSES

Sample Commands:

sent as ASCII text over control channel

USER *username*

PASS *password*

LIST return list of files in current directory

RETR filename retrieves (gets) file

STOR filename stores (puts) file onto remote host

Sample Return Codes:

status code and phrase

331 Username OK, password required

125 data connection already open; transfer starting

425 Can't open data connection

452 Error writing file

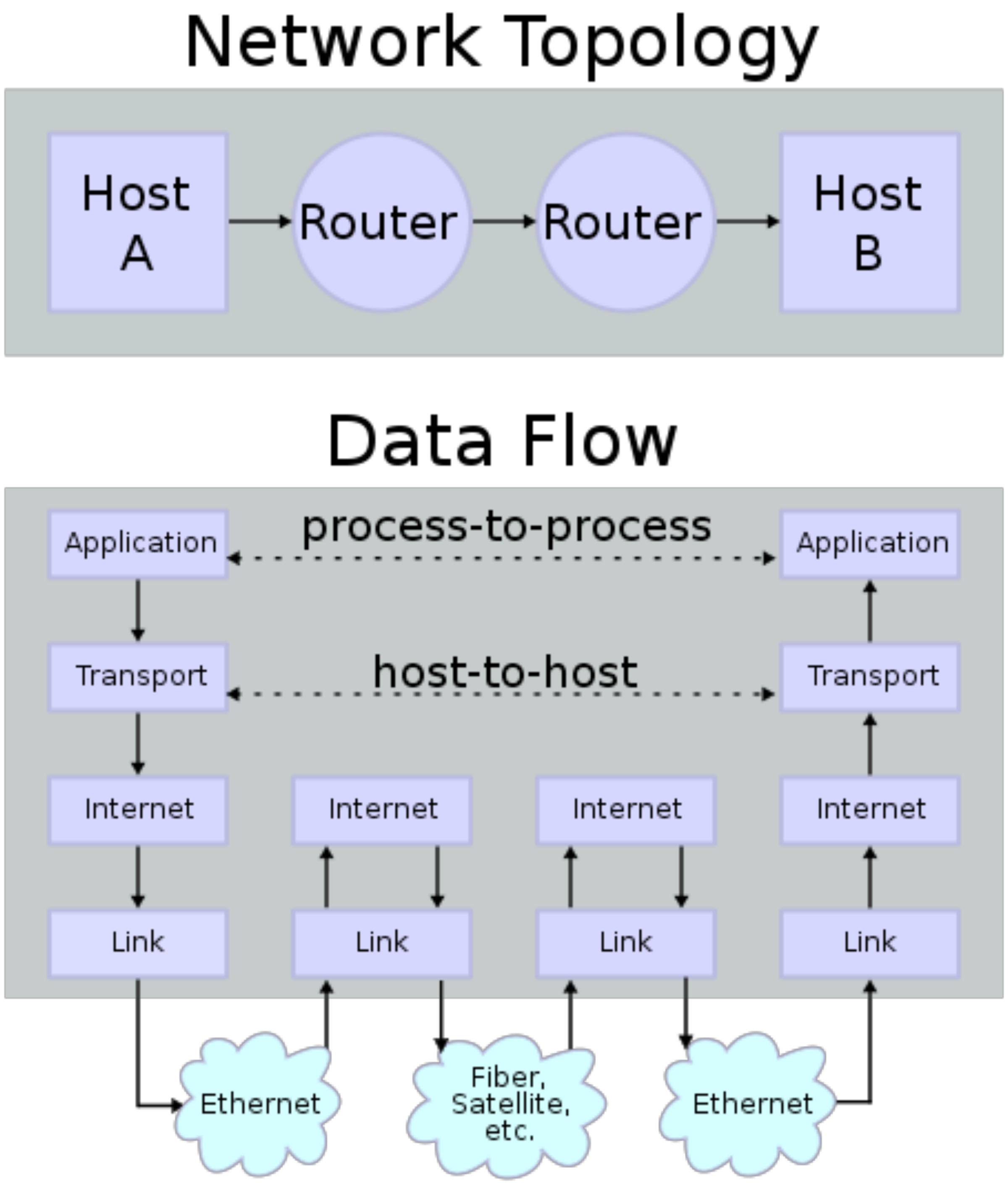
TRANSPORT SERVICE REQUIREMENTS

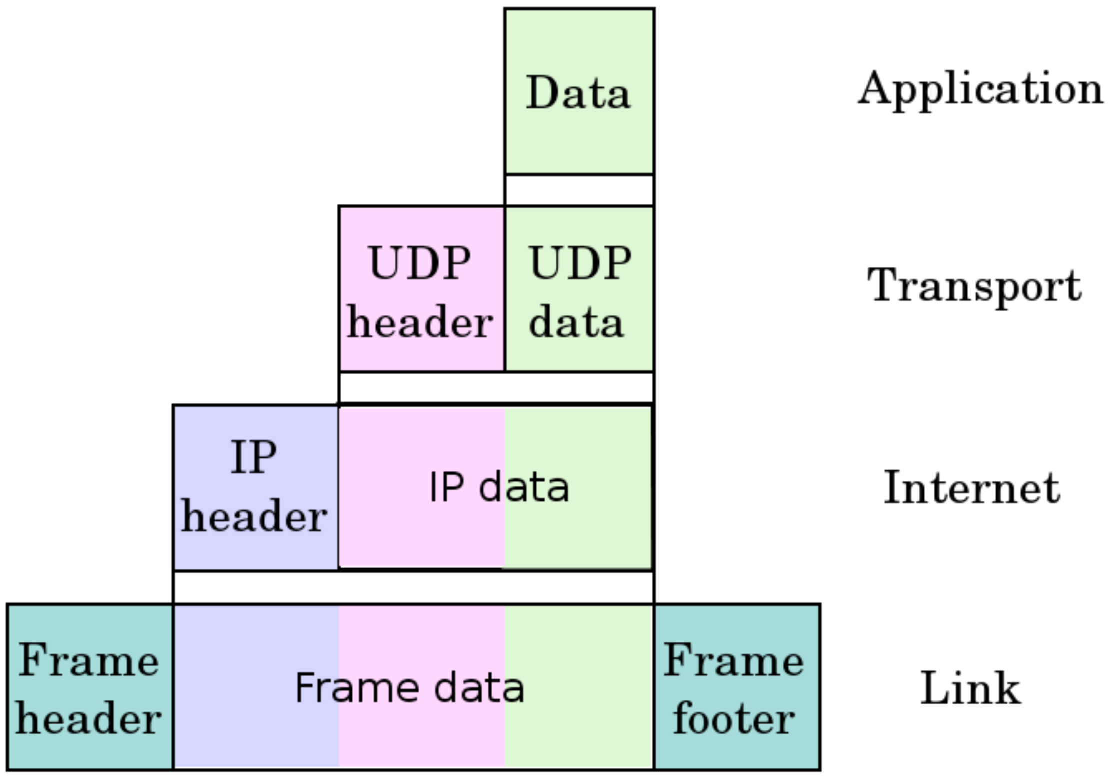
- Data loss
  - Some apps (e.g. audio) can tolerate loss
  - Other apps (e.g. file transfer, telnet) require 100% reliable transfer
- Timing
  - Some apps (e.g. games) require low delay to be effective
- Bandwidth
  - Some apps (e.g. multimedia) require minimum bandwidth to be effective
  - Some apps (e.g. "elastic apps") use whatever bandwidth they can

TRANSPORT SERVICE REQUIREMENTS

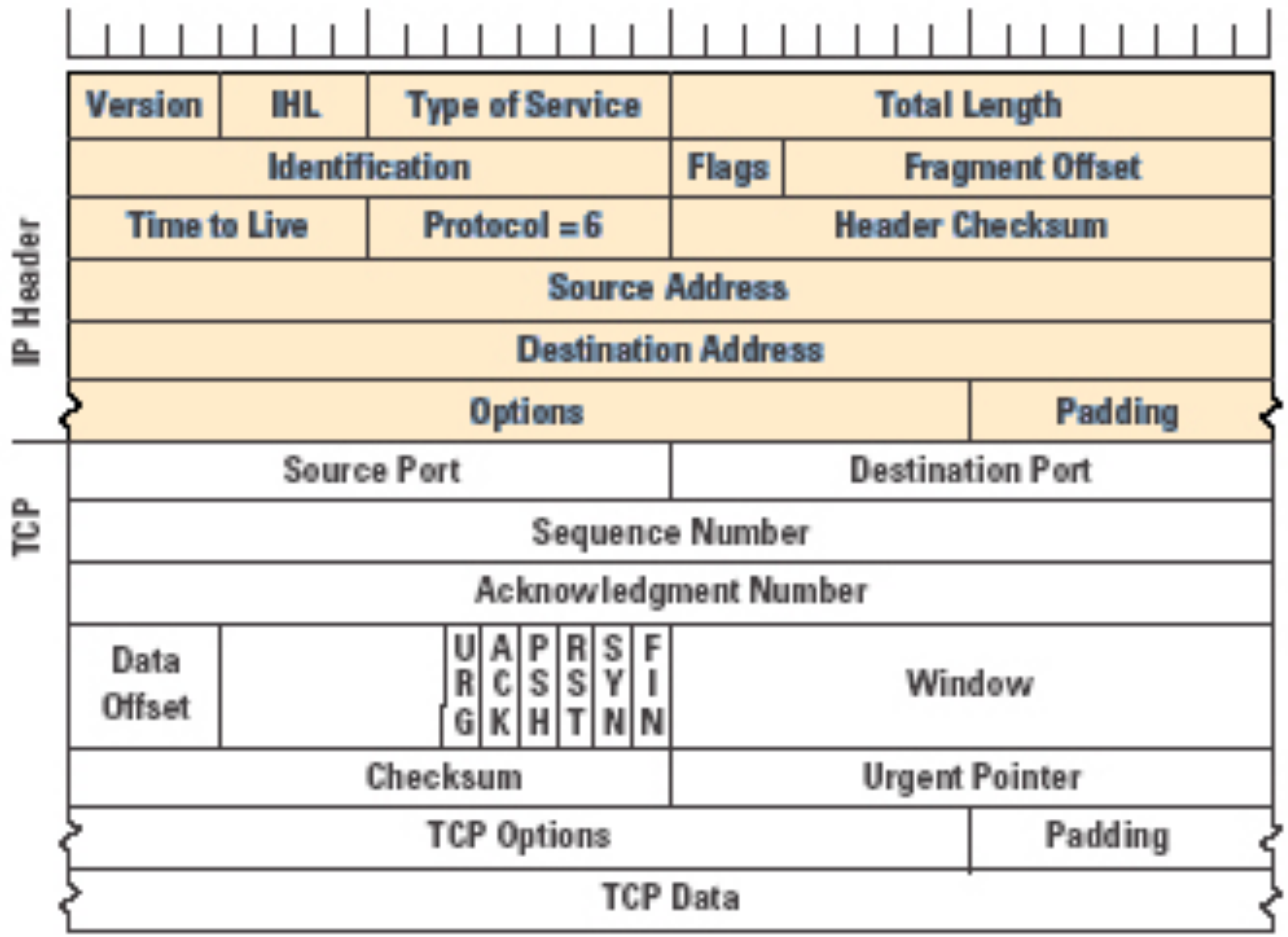
Application	Data loss	Bandwidth	Time Sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
web documents	no loss	elastic	no
real-time audio/ video	loss-tolerant	audio: 5Kb-1Mb video:10Kb-5Mb	yes, 100 msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few Kbps	yes, 100 msec
financial apps	no loss	elastic	yes and no

PACKET FORMAT





PACKET FORMAT

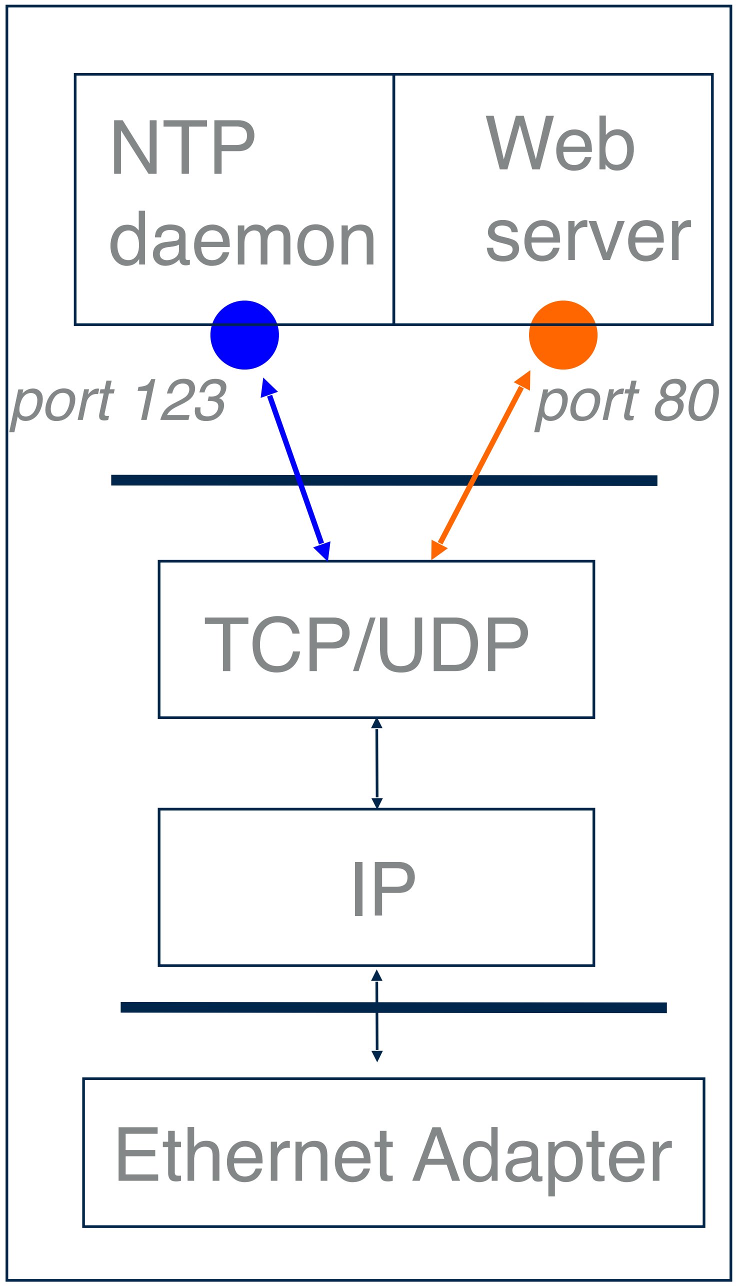


NAMES AND ADDRESSES

- Each attachment point on Internet is given a unique address
  - Based on location within network (like phone numbers)
- Humans prefer to deal with names not addresses
  - Domain Name Service (DNS) provides mapping of name to address
  - Name based on administrative ownership of host

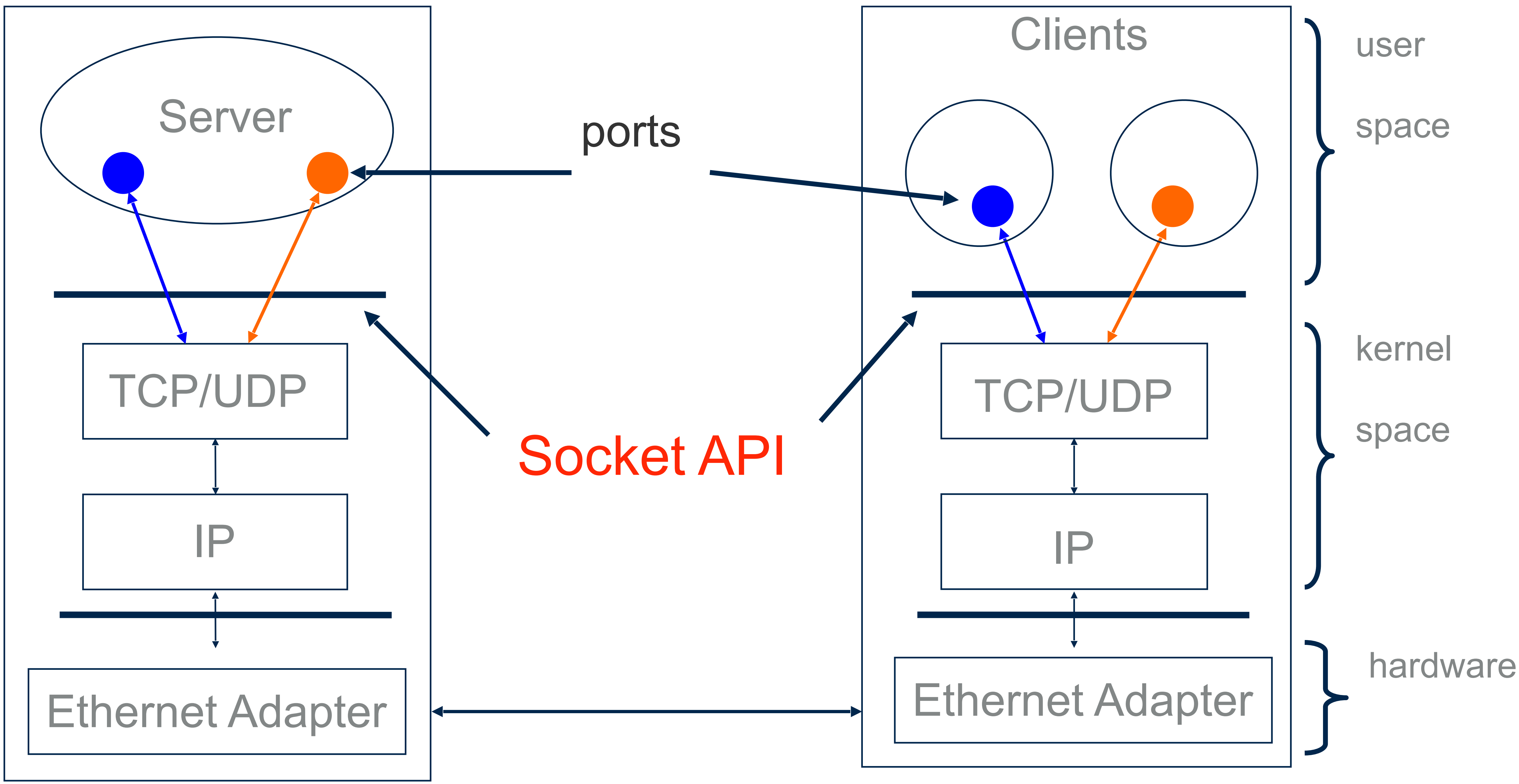
CONCEPT OF PORT NUMBERS

- Port numbers are used to identify "entities" on a host
- Port numbers can be:
  - Well-known (port 0-1023)
  - Assigned (port 1024-49151)
  - Dynamic or private (port 49152-65535)
- Servers/daemons usually use well-known ports
  - Any client can identify the server/service
  - HTTP = 80, FTP = 21, Telnet = 23, ...
- Other common services use assigned ports
- Clients should use dynamic ports
  - Assigned by kernel at runtime



SERVER AND CLIENT

Server and Client exchange messages over the network through a common Socket API



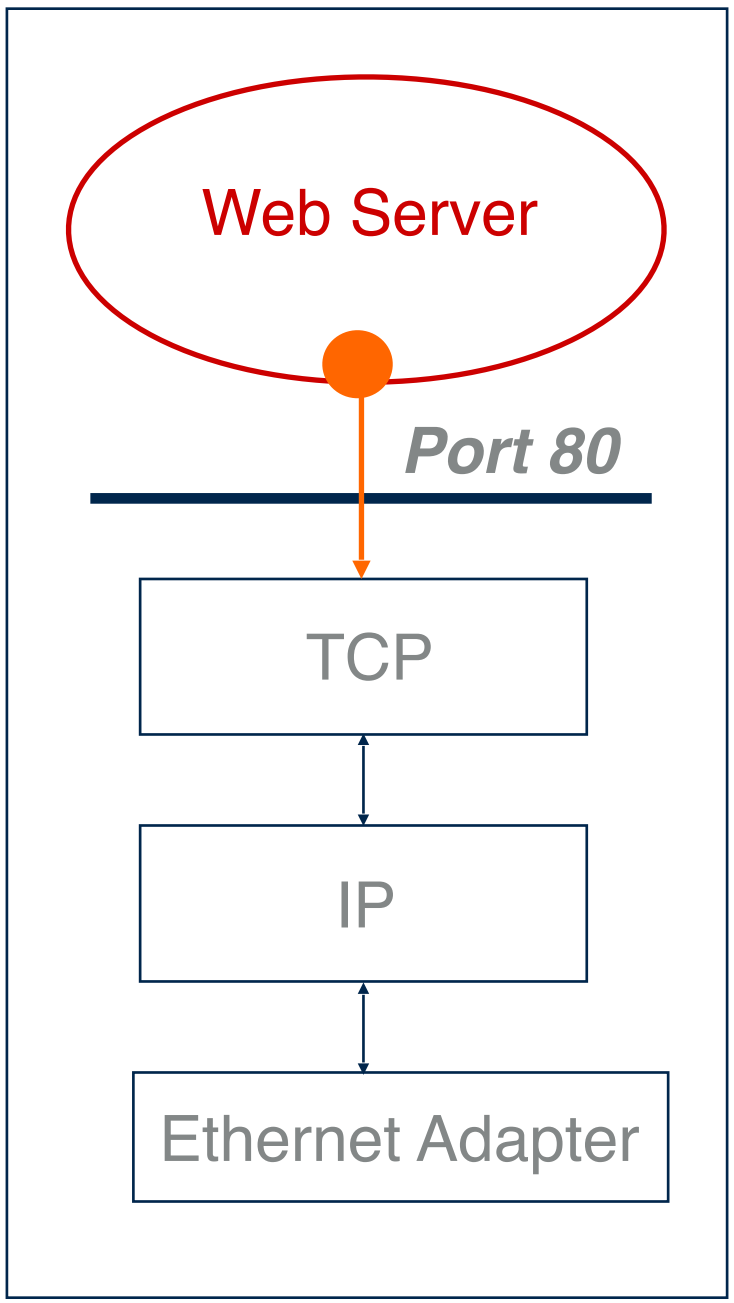
WHAT IS A SOCKET?

- A socket is a **file descriptor** that lets an application read/write data from/to the network

```
int fd;          /* socket descriptor */
if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) }
   perror("socket");
   exit(1);
}
```

- `socket` returns an integer (socket descriptor)
  - fd < 0 indicates that an error occurred
- AF_INET: associates a socket with the Internet protocol family
- SOCK_STREAM: selects the TCP protocol, SOCK_DGRAM: selects the UDP protocol

TCP SERVER



- What does a web server need to do so that a web client can connect to it?

SOCKET I/O: SOCKET()

- Since web traffic uses TCP, the web server must create a socket of type SOCK_STREAM

```
int fd;                /* socket descriptor */


if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
     perror("socket");
     exit(1);
}
```

SOCKET I/O: BIND()

- A **socket** can be bound to a **port**

```
int fd;                              /* socket descriptor */
struct sockaddr_in srv;      /* used by bind() */

/* create the socket */
srv.sin_family = AF_INET; /* use the Internet addr family */
srv.sin_port = htons(80); /* bind socket 'fd' to port 80*/

/* bind: a client may connect to any of my addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
      perror("bind"); exit(1);
}
```

- Still not quite ready to communicate with a client…

SOCKET I/O: LISTEN()

- **listen** indicates that the server will accept a connection

```
int fd;                        /* socket descriptor */
struct sockaddr_in srv;        /* used by bind() */

/* 1) create the socket */
/* 2) bind the socket to a port */

if(listen(fd, 5) < 0) {    /* backlog of 5 */
      perror("listen");
      exit(1);
}
```

  - Still not quite ready to communicate with a client…

SOCKET I/O: ACCEPT()

- **accept** blocks waiting for a connection

```
int fd;                              /* socket descriptor */
struct sockaddr_in srv;             /* used by bind() */
struct sockaddr_in cli;             /* used by accept() */
int newfd;                          /* returned by accept() */
int cli_len = sizeof(cli);          /* used by accept() */
/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */
newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
       perror("accept");    exit(1);
}
```

- **accept** returns a new socket (**newfd**) with the same properties as the original socket (**fd**)
  - **newfd** < 0 indicates that an error occurred

SOCKET I/O: ACCEPT() CONTINUED...

```
struct sockaddr_in cli;            /* used by accept() */
int newfd;                         /* returned by accept() */
int cli_len = sizeof(cli);         /* used by accept() */

newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
        perror("accept");
        exit(1);
}
```

- How does the server know which client it is?
    - `cli.sin_addr.s_addr` contains the client's **IP address**
    - `cli.sin_port` contains the client's **port number**
- Now the server can exchange data with the client using **read** and **write** on the descriptor **newfd**
- Why does **accept** need to return a new descriptor?

SOCKET I/O: READ()

- read *blocks* on data from the client but does not guarantee that sizeof(buf) is read

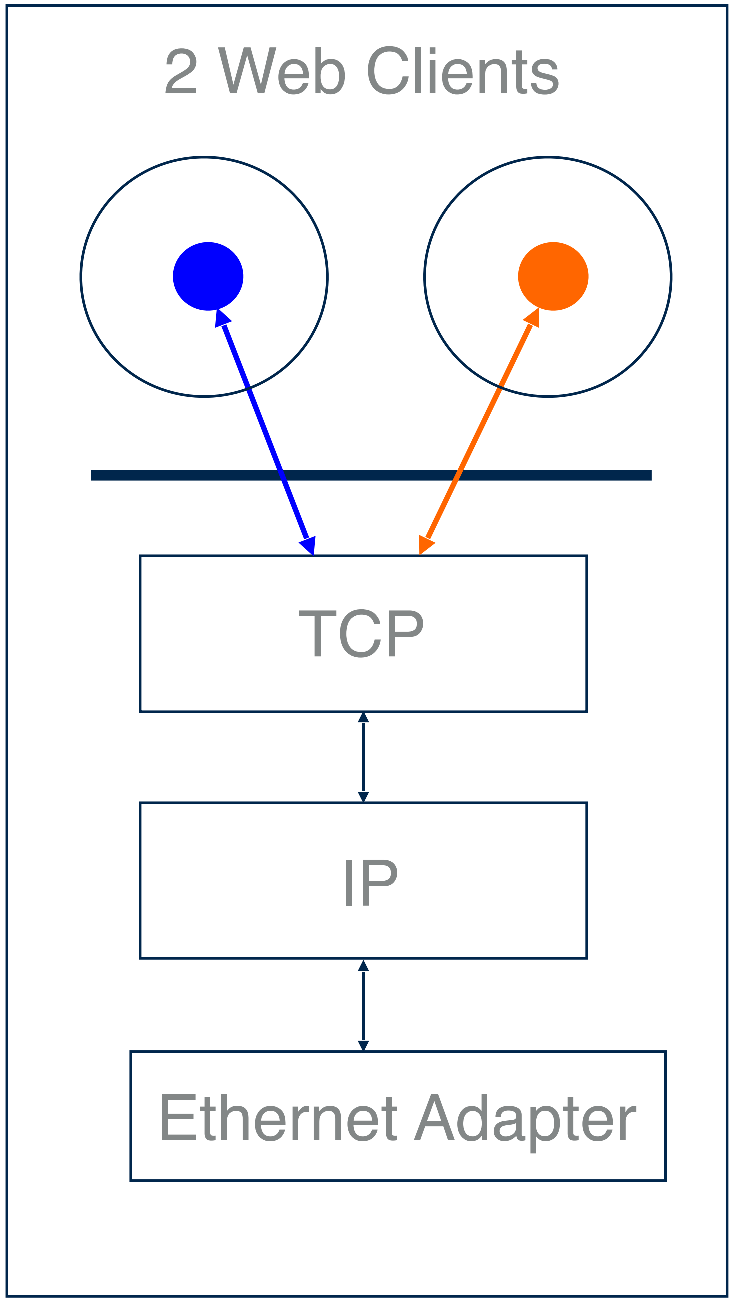
```
int fd;                                    /* socket descriptor */
char buf[512];                             /* used by read() */
int nbytes;                               /* used by read() */

/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */
/* 4) accept the incoming connection */

if((nbytes = read(newfd, buf, sizeof(buf))) < 0) {
       perror("read"); exit(1);
}
```

TCP CLIENT

- How does a web client connect to a web server?



DEALING WITH IP ADDRESSES

- IP Addresses are commonly written as strings (“128.83.144.73”), but programs deal with IP addresses as integers.

Converting strings to numerical address:

```
struct sockaddr_in srv;
srv.sin_addr.s_addr = inet_addr(“128.83.144.73”);
if(srv.sin_addr.s_addr == (in_addr_t) -1) {
      fprintf(stderr, "inet_addr failed!\n"); exit(1);
}
```

Converting a numerical address to a string:

```
struct sockaddr_in srv;
char *t = inet_ntoa(srv.sin_addr);
if(t == 0) {
      fprintf(stderr, “inet_ntoa failed!\n”); exit(1);
}
```

TRANSLATING NAMES TO ADDRESSES

- **getaddrinfo** provides interface to DNS
- Returns `addrinfo` structs given a host and service
- **getnameinfo** provides host and service given `addrinfo`
- Functions are not IPv4 or IPv6 dependent

```
#include <netdb.h>

int st;
struct addrinfo *results; /*ptr to linked list of address info*/
struct addrinfo hints;
char *name = “www.cs.utexas.edu”;
if (st = getaddrinfo(name, “80”, &hints, &results) != 0) {
        fprintf(stderr, “getaddrinfo failed!\n”); exit(1);
}
```

SOCKET I/O: CONNECT()

- **connect** allows a client to connect to a server

```
int fd;                             /* socket descriptor */
struct sockaddr_in srv;             /* used by connect() */

/* create the socket */
/* connect: use the Internet address family */
srv.sin_family = AF_INET;
/* connect: socket ‘fd’ to port 80 */
srv.sin_port = htons(80);
/* connect: connect to IP Address “128.2.35.50” */
srv.sin_addr.s_addr = inet_addr(“128.83.144.73”);

if(connect(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
      perror(”connect"); exit(1);
}
```

SOCKET I/O: WRITE()

- **write** can be used with a socket

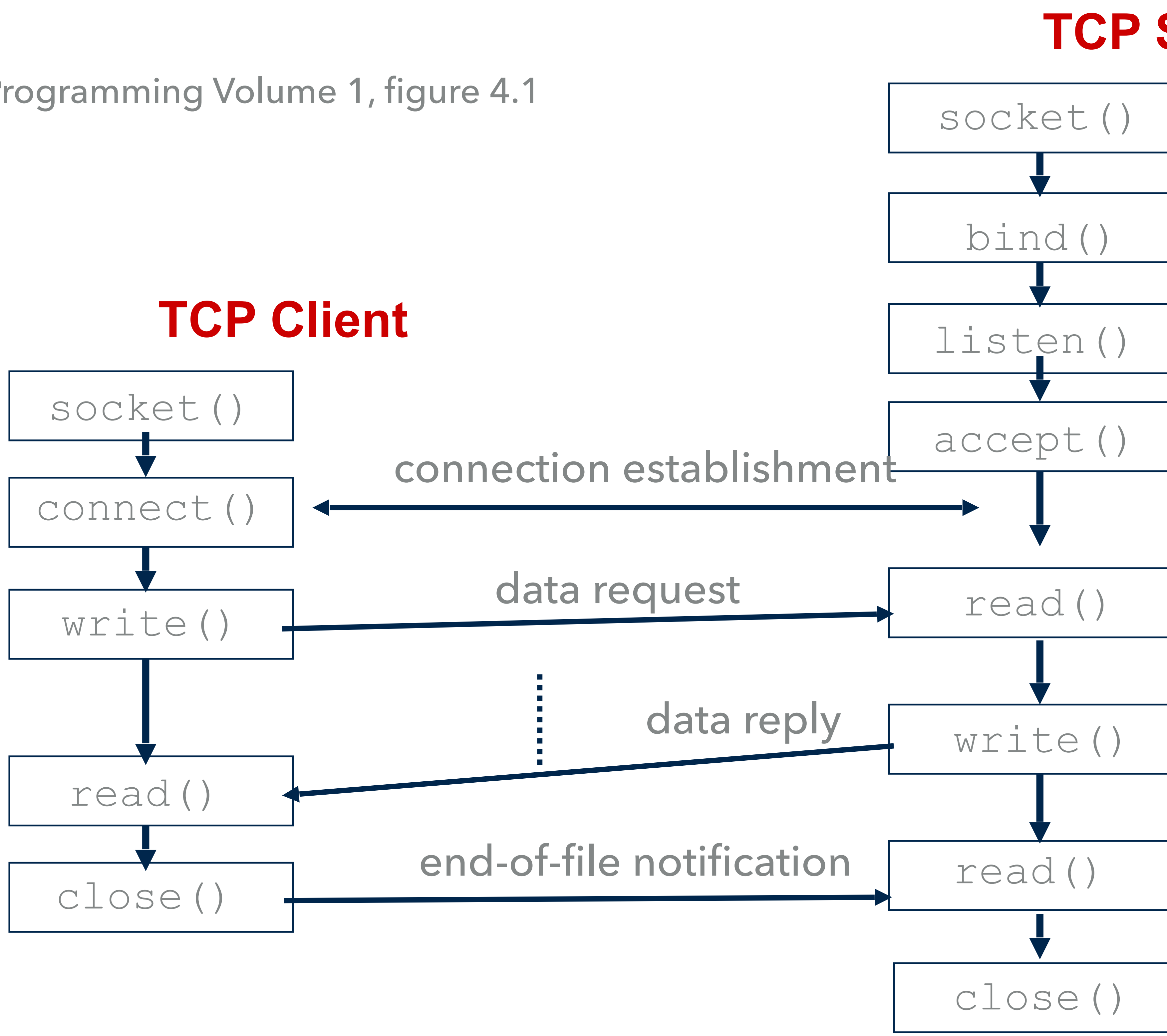
```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by connect() */
char buf[512];                   /* used by write() */
int nbytes;                      /* used by write() */

/* 1) create the socket */
/* 2) connect() to the server */

/* Example: A client could "write" a request to a server */
if((nbytes = write(fd, buf, sizeof(buf))) < 0) {
      perror("write");
      exit(1);
}
```

TCP CLIENT-SERVER INTERACTION

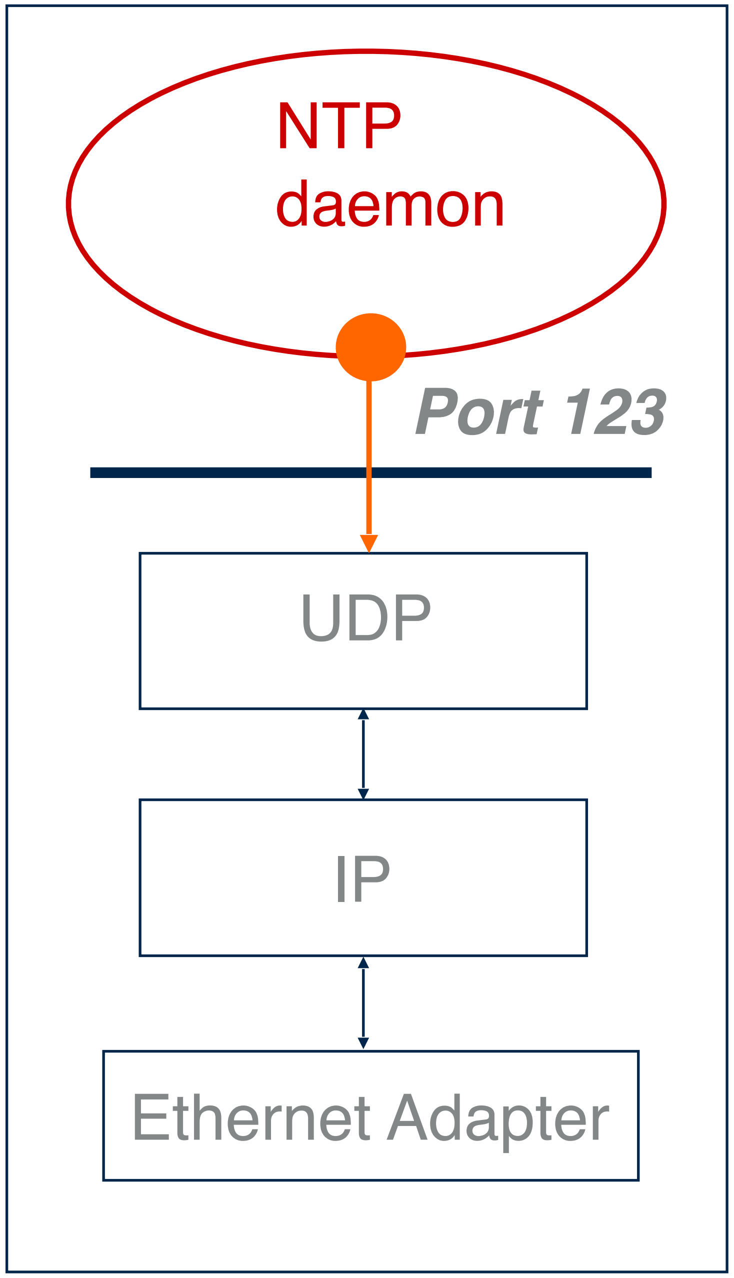
from UNIX Network Programming Volume 1, figure 4.1



UDP PROPERTIES

- Does not assume any handshake or prior communication
- Stateless protocol with no information/session retention
- Uses datagrams or self-contained packets of information
  - No need for prior information exchange

UDP SERVER EXAMPLE



- What does a UDP server need to do so that a UDP client can connect to it?

SOCKET I/O: SOCKET()

- The UDP server must create a datagram socket…

```
int fd;                    /* socket descriptor */

if((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket");
        exit(1);
}
```

- **socket** returns an integer (socket descriptor)
  - **fd** < 0 indicates that an error occurred
- **AF_INET** associates the socket with the Internet protocol family
- **SOCK_DGRAM** selects the UDP protocol

SOCKET I/O: BIND()

- A **socket** can be bound to a **port**

```
int fd;                         /* socket descriptor */
struct sockaddr_in srv;     /* used by bind() */

/* create the socket */
/* bind: use the Internet address family */
srv.sin_family = AF_INET;
/* bind: socket 'fd' to port 80*/
srv.sin_port = htons(80);
/* bind: a client may connect to any of my addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
      perror("bind"); exit(1);
}
```

- Now the UDP server is ready to accept packets…

SOCKET I/O: RECVFROM()

- **read** does not provide the client's address to the UDP server
- **recvfrom** receives messages from a socket

```
int fd;                            /* socket descriptor */
struct sockaddr_in srv;            /* used by bind() */
struct sockaddr_in cli;            /* used by recvfrom() */
char buf[512];                     /* used by recvfrom() */
int cli_len = sizeof(cli);         /* used by recvfrom() */
int nbytes;                        /* used by recvfrom() */
/* 1) create the socket */
/* 2) bind to the socket */
nbytes = recvfrom(fd, buf, sizeof(buf), 0 /* flags */,
               (struct sockaddr*) &cli, &cli_len);
if(nbytes < 0) {
      perror("recvfrom"); exit(1);
}
```

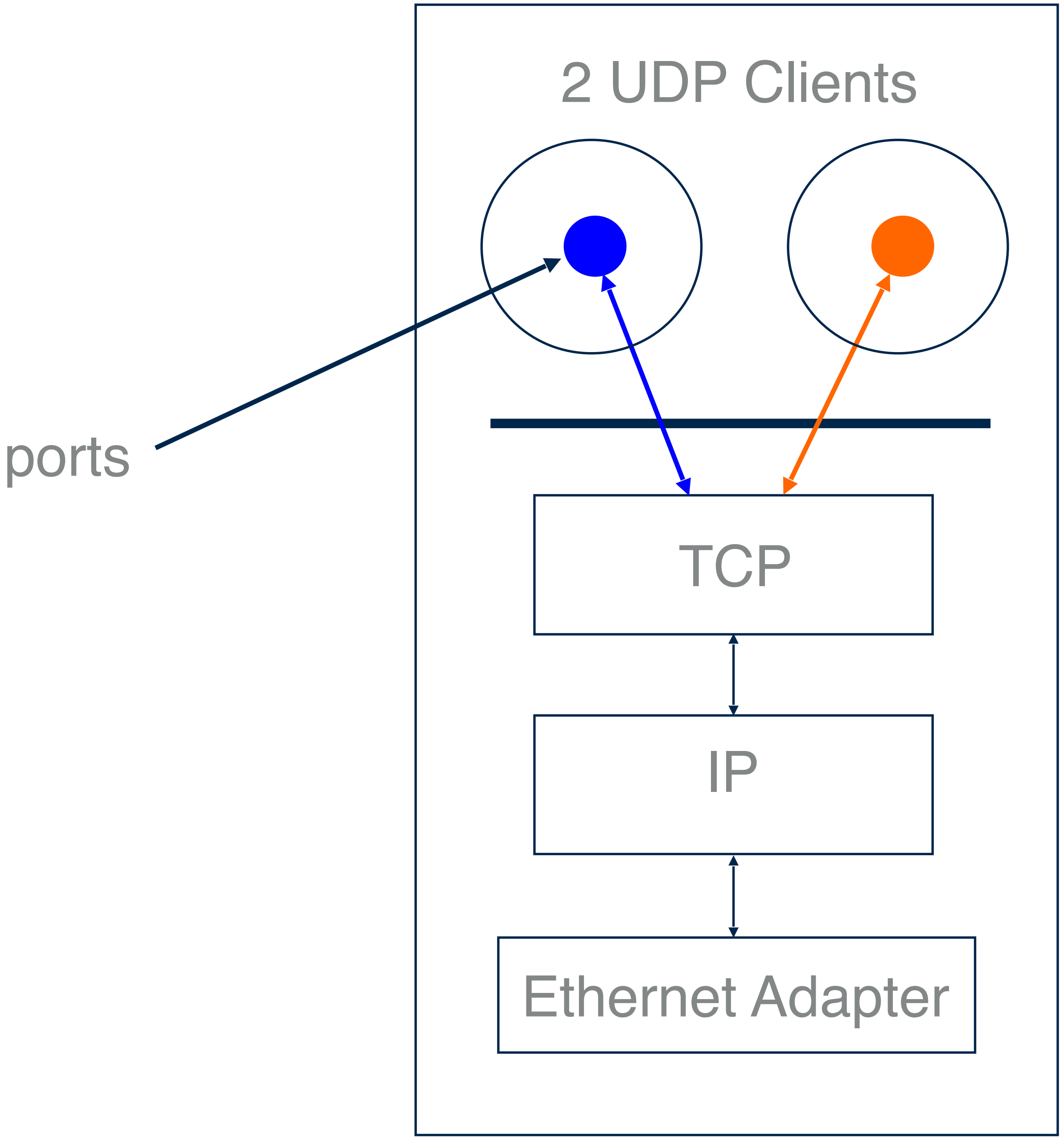
SOCKET I/O: RECVFROM() CONTINUED…

- The actions performed by **recvfrom**
  - Returns the number of bytes to read (**nbytes**)
  - Copies nbytes of data into **buf**
  - Returns the address of the client (**cli**)
  - Returns the length of cli (**cli_len**)

```
nbytes = recvfrom(fd, buf, sizeof(buf), 0 /* flags */,
                  (struct sockaddr*) cli, &cli_len);
```

UDP CLIENT EXAMPLE

- How does a UDP client communicate with a UDP server?



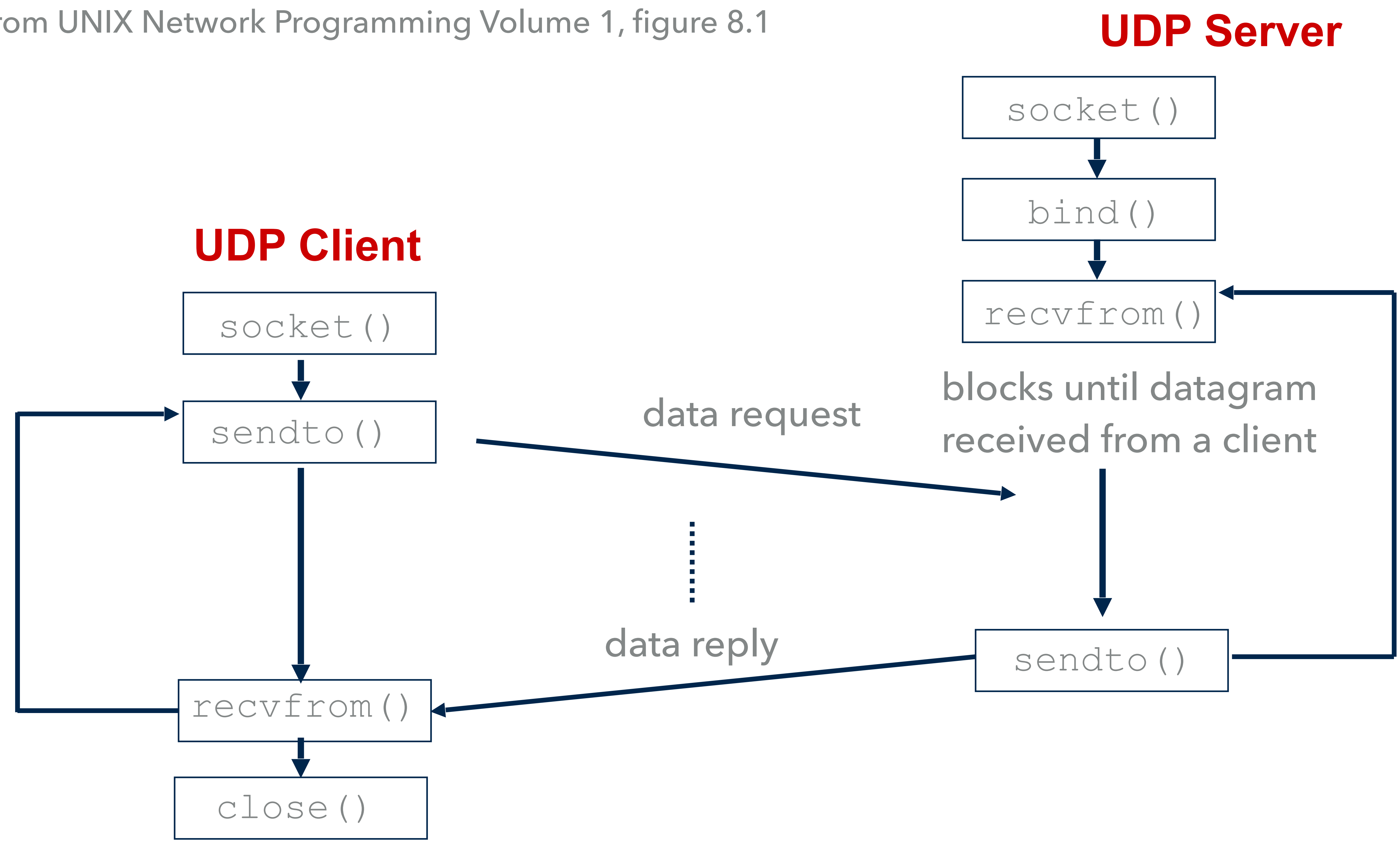
SOCKET I/O: SENDTO()

- **write** is not allowed
- UDP client does not **bind** a port number
  - Port number is dynamically assigned when the first sendto is called

```
int fd;                               /* socket descriptor */
struct sockaddr_in srv;               /* used by sendto() */
/* 1) create the socket */
/* sendto: send data to IP Address "128.2.35.50" port 80 */
srv.sin_family = AF_INET;
srv.sin_port = htons(80);
srv.sin_addr.s_addr = inet_addr("128.83.144.73");
nbytes = sendto(fd, buf, sizeof(buf), 0 /* flags */,
                 (struct sockaddr*) &srv, sizeof(srv));
if(nbytes < 0) {
       perror("sendto");     exit(1);
}
```

UDP CLIENT-SERVER INTERACTION

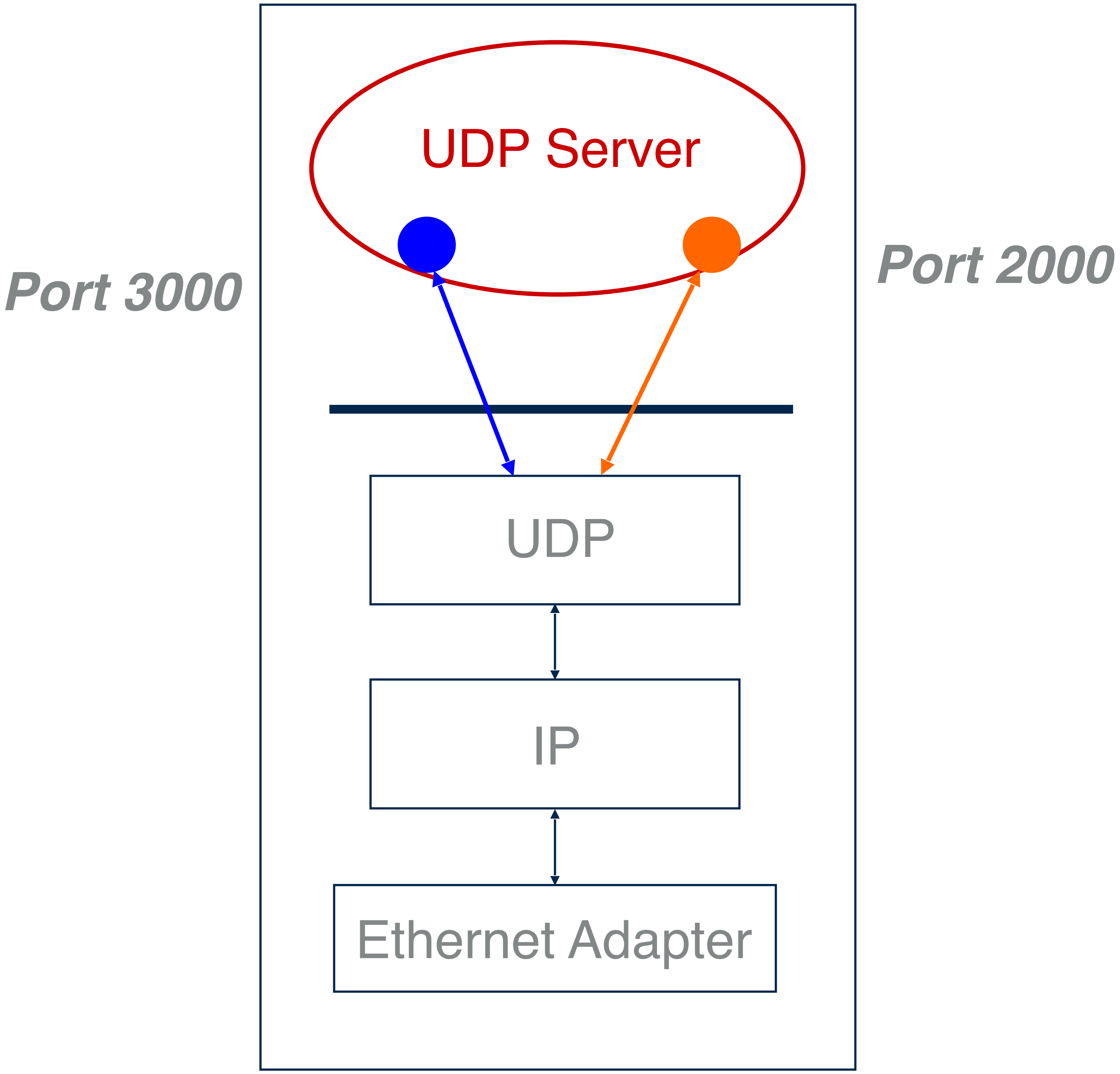
from UNIX Network Programming Volume 1, figure 8.1



SIDE NOTE: UDP BROADCAST AND MULTICAST

- These examples have been point-to-point (one source, one destination) sending of data but UDP supports point-to-multipoint (one source, multiple destinations)
- May not work in all circumstances and primarily for LANs
  - Broadcast only supported in IPV4
  - Multicast not supported by all switches and hubs
  - Only way to do it across the Internet is with additional work-arounds
- IP Multicast added to IPV4 and fully integrated in IPV6
  - Primarily for multimedia content

THE UDP SERVER



- How can the UDP server service multiple ports simultaneously?

UDP SERVER: SERVICING TWO PORTS

- What problems does this code have?

```
int s1;                          /* socket descriptor 1 */
int s2;                          /* socket descriptor 2 */

/* 1) create socket s1 */
/* 2) create socket s2 */
/* 3) bind s1 to port 2000 */
/* 4) bind s2 to port 3000 */

while(1) {
      recvfrom(s1, buf, sizeof(buf), ...);
      /* process buf */

      recvfrom(s2, buf, sizeof(buf), ...);
      /* process buf */
}
```

SOCKET I/O: SELECT()

```
int select(int maxfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timeval *timeout);

FD_CLR(int fd, fd_set *fds);    /* clear the bit for fd in fds */
FD_ISSET(int fd, fd_set *fds); /* is the bit for fd in fds? */
FD_SET(int fd, fd_set *fds);    /* turn on the bit for fd in fds */
FD_ZERO(fd_set *fds);           /* clear all bits in fds */
```

- **maxfds**: number of descriptors to be tested
  - descriptors (0, 1, ... maxfds-1) will be tested
- **readfds**: a set of *fds* we want to check if data is available
  - returns a set of *fds* ready to read
  - if input argument is *NULL*, not interested in that condition
- **writefds**: returns a set of *fds* ready to write
- **exceptfds**: returns a set of *fds* with exception conditions

SOCKET I/O: SELECT()

```
int select(int maxfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

struct timeval {
      long tv_sec;          /* seconds /
      long tv_usec;               /* microseconds */
}
```

- **timeout**
  - if NULL, wait forever and return only when one of the descriptors is ready for I/O
  - otherwise, wait up to a fixed amount of time specified by timeout
    - if we don't want to wait at all, create a timeout structure with timer value equal to 0

SOCKET I/O: SELECT() IN UDP

- **select** allows synchronous I/O multiplexing

```
int s1, s2;                        /* socket descriptors */
fd_set readfds;            /* used by select() */

/* create and bind s1 and s2 */
while(1) {
        FD_ZERO(&readfds);                  /* initialize the fd set */
        FD_SET(s1, &readfds);     /* add s1 to the fd set */
        FD_SET(s2, &readfds);     /* add s2 to the fd set */

        if(select(s2+1, &readfds, 0, 0, 0) < 0) {
                perror("select");
                exit(1);
        }
        if(FD_ISSET(s1, &readfds)) {
                recvfrom(s1, buf, sizeof(buf), ...);
                /* process buf */
        }
        /* do the same for s2 */
}
```

SOCKET I/O: SELECT() IN TCP

```
int fd, next=0;                                  /* original socket */
int newfd[10];                              /* new socket descriptors */
while(1) {
       fd_set readfds;
       FD_ZERO(&readfds);
       FD_SET(fd, &readfds);

       /* Now use FD_SET to initialize other newfd's
          that have already been returned by accept() */

       select(maxfd+1, &readfds, 0, 0, 0);
       if(FD_ISSET(fd, &readfds)) {
              newfd[next++] = accept(fd, ...);
       }
       /* do the following for each descriptor newfd[n] */
       if(FD_ISSET(newfd[n], &readfds)) {
              read(newfd[n], buf, sizeof(buf));
              /* process data */
       }
}
```

EVENT-DRIVEN APPROACHES

- Use of asynchronous event notifications
  - Potentially faster and more flexible than select
- Provide notifications when events occur on file descriptors
- Designed to handle event loop in a fast, non-blocking way
- Libraries like libevent, libev, libuv, etc

BASIC PACKET BUILDING FOR A BUFFER

```
struct packet {
      u_int32_t type;
      u_int16_t length;
      u_int16_t checksum;
      u_int32_t address;
};

/* ===================================================== */
char buf[1024];
struct packet *pkt;

pkt = (struct packet*) buf;
pkt->type = htonl(1);
pkt->length = htons(2);
pkt->checksum = htons(3);
pkt->address = htonl(4);
```

EXTENDING FUNCTIONALITY THROUGH PACKETS

- Possible to use TCP and UDP to get functionality of both protocols
- Also possible to add packet information and packet handling to UDP communication for greater reliability
  - e.g. Index checks on packets to verify order and delivery
- System needs and constraints determine how to approach problem
  - Don't reinvent TCP
  - But maybe a little more reliability is worth latency tradeoffs…

PAYLOAD CONSIDERATIONS

- What information needs to be in the packet?
- How large is the payload?
- What is the latency of serializing/deserializing the payload?
- How often do the server and clients need to know about this information?
- Is my payload secure and safe?

PACKET INFORMATION

- What information is in what packet should be architected with care
  - Cannot afford to send out the entire world state every frame
- Provide initial information about world schema to client upon connection
- Provide ongoing updates relative to this schema as the world state changes

DISCUSS

- Consider these client-server network scenarios. What should be in the packet? What needs to happen when the packet is received?
  - A player in an MMO trades with another player
  - A player in a battle royale equips a new weapon
  - A player in a go game places a stone
  - A player in an arena shooter uses a hit scan gun
  - A player in an arena shooter uses a ballistic gun

PACKET FORMAT

- XML and JSON are too verbose for the frequency data is being sent
- Text information is not tightly packed
- Ideally use a binary format
  - Low latency games may use a custom binary format rather than an existing library

PROBLEMS WITH MEMCPY

- Directly copying the struct data into the packet is very cheap
  - Works well on simple projects like what we're creating where only 4 or 5 people will play it
- Major issues at a commercial level
  - Must ensure cross-platform/cross-compiler support for memory layout
  - Must handle endian-ness
  - Must handle pointers
  - Major security risk if struct data is simply trusted

READING AND WRITING PER-FIELD

- Create serialization library that reads and writes from the struct to the packet
  - Need to be able to read/write from every struct type
  - Need to be able to read/write into every packet type
- Can additionally perform better bitpacking here to ensure good packet properties

ADDITIONAL RESOURCES

- Gaffer on Games <https://www.gafferongames.com/>
  - Tons of in-depth articles on physics, networking, and networked physics