Programming in C

Socket Programing

Reference Material

- User friendly introduction to networking http://www.ecst.csuchico.edu/~beej/guide/net/html
- Unix Network Programming, W. R. Stevens
  - book code: http://www.kohala.com/start/unpv12e.html
- The GNU C library:
  - http://www.cs.utah.edu/dept/old/texinfo/glibc-manual-0.02/library_toc.html

Hardware Addresses

- Hosts access the physical medium via network cards.
- Each network card is uniquely identified by a 48 bit (6 bytes) number, called hardware address, or Ethernet address.
- Ethernet addresses are hardwired into the electronics of the network device.

$$\underbrace{b_1\ b_2\ b_3}\ \underbrace{b_4\ b_5\ b_6}$$

unique to the manufacturer of the card.

assigned by the manufacturer.

- ARP/RARP protocols map IP addresses to hardware addresses and vice versa.

IP Addresses

- Hosts are identified in the network by 32-bit IP addresses (xxx.xxx.xxx.xxx) (also referred as IPv4 addresses).
- Each decimal number represents eight bits of binary data (value between 0 and 255).
- Divided in classes.
- Examples:
  - 128.220.224.76

IPv4 vs. IPv6 Address

- Two different network addresses:
  - IPv4 addresses: 32 bits addresses.
  - IPv6 addresses: 128 bits addresses.
- IPv4 is still the one mostly used. Many of the networking functions handle both network addresses families.
- The examples in this lecture will use IPv4.

Hostnames and IP Addresses

- People prefer names for hosts (hostnames):
  - Name: ugrad1
  - Fully qualified name: ugrad1.cs.jhu.edu
- DNS (Domain Name System) maps hostnames to IP addresses.
- Example:

  ugrad1.cs.jhu.edu has the IP 128.220.224.76

Ports

- Remember:
  - Hardware addresses identity network cards
  - IP addresses identify hosts
  - Names identify hosts in a human friendly way.
- However, transport protocols (TCP and UDP) ensure communication between processes.
- How do computers differentiate what data is for which process?
- Port numbers

  16-bit numbers

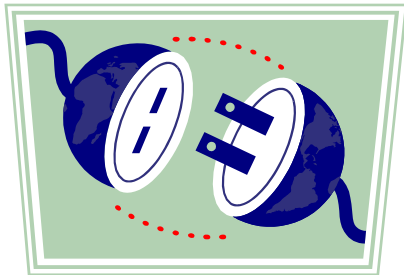
Ports contd.

- In general servers use well-known ports, while clients use ephemeral ports
- Example: port 80 is assigned to web server (HTTP)
- Port numbers:
  - Well-known ports: 0 - 1023
  - Registered ports: 1024 – 49151
  - Dynamic/private ports: 49152 - 65535

Socket and Socket Pair

- Socket: identifies a communication end-point.

  socket = (IP address, port number)



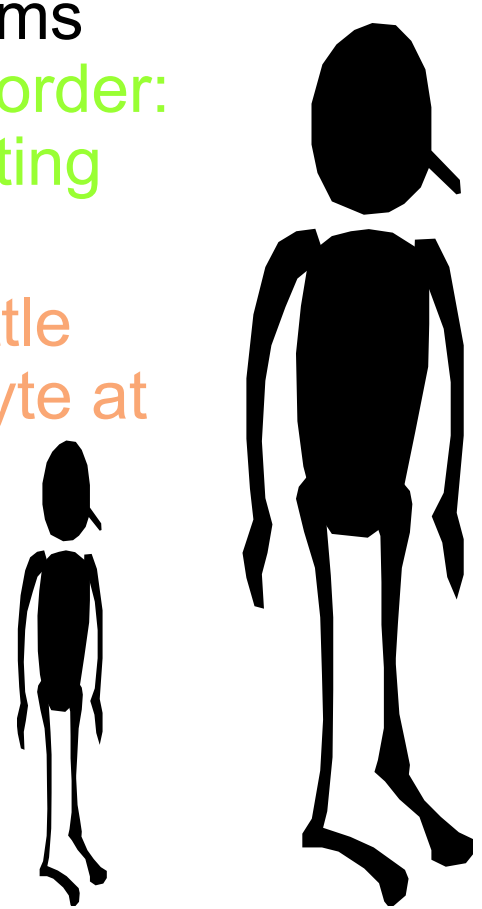
- Socket pair: uniquely identifies a TCP connection over the Internet:

  socket pair = (local IP address, local IP port,
  remote IP, remote IP port)

- Socket pair concept can also be extended to UDP.

Byte Order

- Different systems store multibyte values (for example int) in different ways.
  - HP, Motorola 68000, and SUN systems store multibyte values in Big Endian order: stores the high-order byte at the starting address
  - Intel 80x86 systems store them in Little Endian order: stores the low-order byte at the starting address.
- Why is this a problem for network applications? Data is interpreted differently on different host.



Byte Order Conversions

- By convention, network order is Big Endian.
- Set of functions for byte order conversion:

```
#include <netinet/in.h>
unsigned long  int htonl(unsigned long int hlong);
unsigned short int htons(unsigned short int hshort);
unsigned long  int ntohl(unsigned long int nlong);
unsigned short int ntohs(unsigned short int nshort);
```

- h means for host
- n means for network
- l means for long integer
- s means short integer

Name and Address Conversions

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);

#include <sys/socket.h>          /* for AF_INET */
struct hostent *gethostbyaddr(const char *addr,
      int len, int type);

 struct hostent {
   char    *h_name;    /* official name of host */
   char    **h_aliases;    /* alias list */
   int     h_addrtype;     /* host address type */
   int     h_length;       /* length of address */
   char    **h_addr_list;  /* list of addresses */
                           /* in network byte order. */
 }
```

hostent

```
struct hostent {
    char    *h_name;    /* official name of host */
    char    **h_aliases;    /* alias list */
    int     h_addrtype;     /* host address type */
    int     h_length;       /* length of address */
    char    **h_addr_list;  /* list of addresses */
 }
```

- `h_name`: the official name of the host.
- `h_aliases`: a zero-terminated array of alternative names for the host.
- `h_addrtype`: type of address; always `AF_INET` at present.
- `h_length`: length of the address in bytes.
- `h_addr_list`: a zero-terminated array of network addresses for the host in network byte order.

h_errno

- The variable `h_errno` can have the following values:
  - `HOST_NOT_FOUND`: the specified host is unknown.
  - `NO_ADDRESS or NO_DATA`: the requested name is valid but does not have an IP address.
  - `NO_RECOVERY`: A non-recoverable name server error occurred.
  - `TRY_AGAIN`: A temporary error occurred on an authoritative name server. Try again later.

gethostbyname

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
```

- Returns a structure of type `hostent` for the given host name. The argument name is either a host name, an IPv4 address or an IPv6 address.
- On failure, it returns a NULL pointer and the error code `h_errno` is set appropriately. The `herror` function can be used to print the error message describing the failure.

Example

```
int main(int argc, char*argv[]) {
  struct hostent *h=NULL;

  if(argv[1] == NULL) {
    fprintf(stderr, “Provide hostname\n”);
    exit(1);
   }
  h = gethostbyname(argv[1]);
  if(h == NULL) {
      herror(“ss”);
      exit(1);
  }
  printf("%s \n", h->h_name);
  return 0;
}
```

gethostbyaddr

```
#include <sys/socket.h>
struct hostent *gethostbyaddr(const char *addr,
          int len, int type);
```

- Returns a structure of type `hostent` for the given host address `addr` of length `len` and address type `type`. The only valid address type is currently `AF_INET`.
- On failure, it returns a NULL pointer and the error code `h_errno` is set appropriately. The `herror` function can be used to print an error message describing the failure.

More Conversions

```
#include <arpa/inet.h>
int inet_aton(const char *str, struct_in
  addr *addr);
in_addr_t inet_addr(const char *str);
char *inet_ntoa(struct in_addr in);
```

- These functions make conversions from of IP addresses from string to binary byte order or vice versa. They are deprecated. Use inet_ntop and inet_pton instead.

inet_ntop

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
const char *inet_ntop(int af, const void *src, char *dst,
   size_t cnt);
```

- Converts the network address structure `src` in the `af` address family into a string in `dst`, which is `cnt` bytes long. Address families are currently supported:
  - `AF_INET`: `src` points to a struct `in_addr` (network byte order format) which is converted to an IPv4 network address. `Dst` must be at least `INET_ADDRSTRLEN` bytes long.
  - `AF_INET6`: `src` points to a struct `in6_addr` (network byte order format) which is converted to an IPv6 network address. `Dst` must be at least `INET6_ADDRSTRLEN` bytes long.

inet_pton

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
int inet_pton(int af, const char *src, void
   *dst);
```

- Converts the character string `src` into a network address structure in the `af` address family, then copies the network address structure to `dst`.
- The following address families are currently supported:
  - `AF_INET`: `src` points to IPv4 network address
  - `AF_INET6`: `src` points IPv6 network address

Example

```
struct hostent *h;
char *p, **pp;
char str[INET_ADDRSTRLEN];

h = gethostbyname(“commedia");
if(h == NULL) {
  herror(“ss”);
  exit(1);
}
printf("%s \n", h->h_name);
for(pp=h->h_aliases; *pp != NULL; pp++) {
  printf("\talias: %s\n", *pp);
}

if(h->h_addrtype == AF_INET) {
  for(pp=h->h_addr_list; *pp != NULL; pp++) {
    printf("\tadresses: %s\n",
       inet_ntop(h->h_addrtype, *pp, str,  sizeof(str)));
  }
}
```

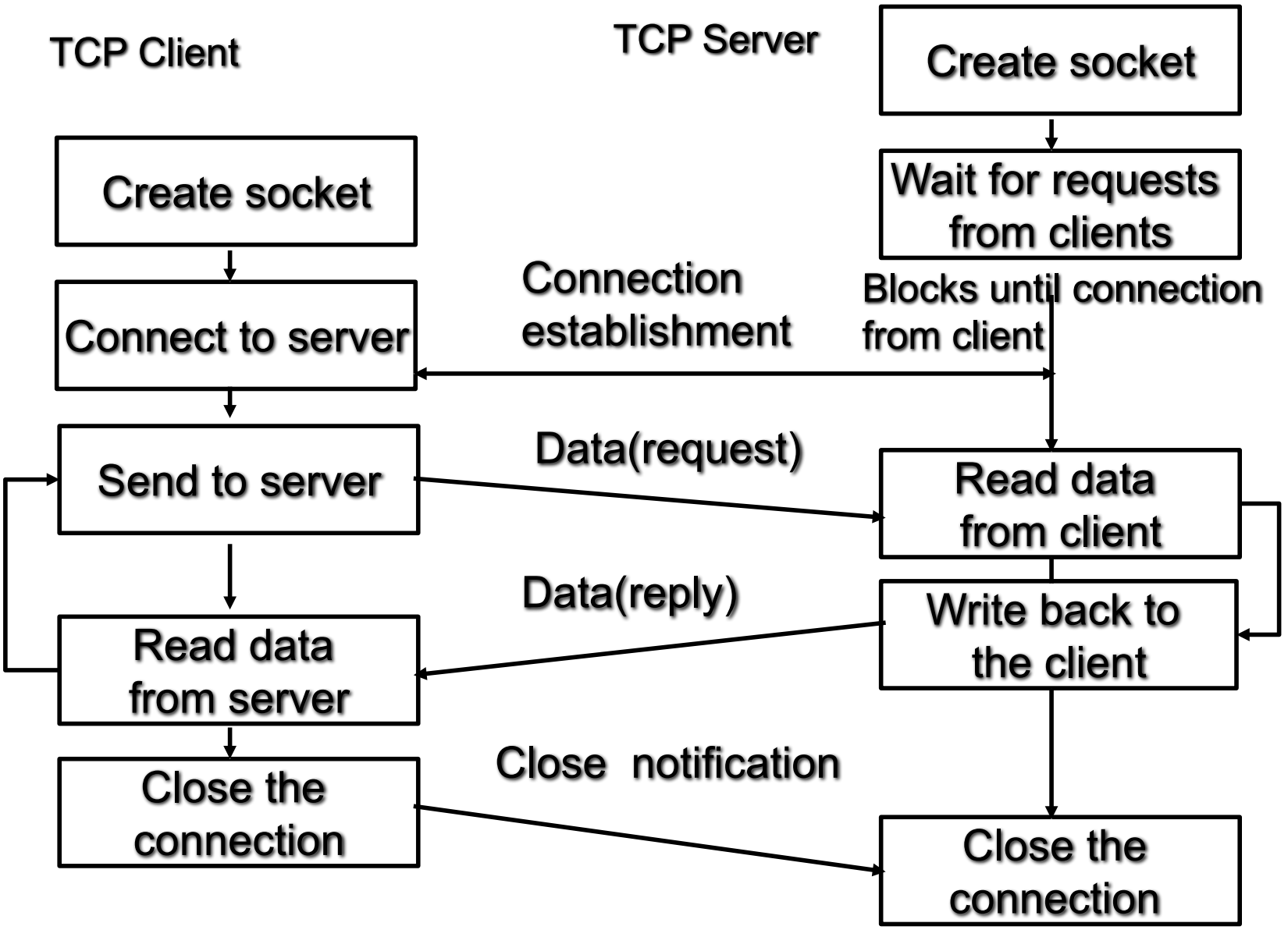
IPv4 Socket Address Structure

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

Simple TCP Client-Server

- Server
  - connects to a well-known port and waits for client connections
  - For each client, prints on the screen what the client send him, and send the data back.
- Client
  - Connects to the well-known port of the server and sends him data
  - Prints on the screen whatever he received from the server

Simple Client-Server TCP Example



Library support

- socket
- Bind, getsockname, getpeername
- Connect
- Listen
- Accept
- Close
- Select

IPv4 Socket Address Structure

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

IPv6 Socket Address Structure

```
struct in6_addr {
  uint8_t_t s6_addr[16];  /* 128 bit IPv6 address
                             network byte order */
}
struct sockaddr_in6 {
  uint8_t         sin6_len;     /* length of struct.*/
  sa_family_t     sin6_family; /* AF_INET6 */
  in_port_t       sin6_port;    /* 16-bit TCP/UDP port*/
                                 /* network byte order */
  uint32_t        sin6_flowinfo; /* priority & flow label,
                                    network byte order */
  struct in6_addr sin6_addr;    /* IPv6 address
                                   network byte order */
}
```

Generic Socket Address Structure

- All networking functions take socket arguments by reference and also need to handle both socket structure types.
- Solution: all take reference to a generic socket structure.

```
struct sockaddr {
  uint8_t         sin_len;    /* length of struct.*/
  sa_family_t     sin_family; /* AF_xxx */
  char            sa_data[14]; /*protocol specific*/;
}
```

socket

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

- Creates an endpoint for communication and returns a descriptor.
- On success returns a descriptor referencing the socket. On failure returns -1 and sets `errno` appropriately.

socket contd.

```
int socket(int domain, int type, int protocol);
```

- `domain` : protocol communication family. Examples:
  - PF_INET IPv4 Internet protocols
  - PF_INET6 IPv6 Internet protocols
- `type`: communication semantics. Examples:
  - SOCK_STREAM: Provides sequenced, reliable, two-way, connection- based byte streams. An out-of-band data transmission mechanism may be supported.
  - SOCK_DGRAM Supports datagrams (connectionless, unreliable messages of a fixed maximum length).
- `protocol`: a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family.

socket contd.

- Examples:

```
/*  Create a TCP socket */
 tcpfd = socket(PF_INET, SOCK_STREAM, 0)
```

```
 /* Create an UDP socket */
udpfd = socket(PF_INET, SOCK_DGRAM, 0);
```

connect

```
#include <sys/socket.h>
int connect(int s, const struct sockaddr *address,
   socklen_t address_len);
```

- Requests a connection to be made on a socket .
  - `s`: the file descriptor associated with the socket.
  - `address`: contains the peer address.
  - `address_len`: the length of the `sockaddr` structure pointed to by the `address` argument.
- On success, returns 0. On failure, returns -1 and `errno` is set to indicate the error.
- If the socket has not already been bound to a local address, connect will bind it: choose and ephemeral port and the IP address if needed.

connect contd.

```
int connect(int s, const struct sockaddr *address,
   socklen_t address_len);
```

- UDP (SOCK_DGRAM sockets):
  - `connect` sets the socket's peer address, but no connection is made. The peer address identifies where all datagrams are sent on subsequent `send`, and limits the remote sender for subsequent `recv` calls.
- TCP (SOCK_STREAM):
  - `connect` attempts to establish a connection to the address `address`. The connection can be established synchronously (connect will block), or asynchronously (`select` will indicate when the file descriptor for the socket is ready for writing).
- If `connect` fails, the state of the socket is unspecified. Close the file descriptor and create a new socket before attempting to reconnect.

bind

```
#include <sys/types.h>
#include <sys/socket.h>
int  bind(int  sockfd, struct sockaddr *my_address,
   socklen_t addrlen);
```

- Assigns an address to an unnamed socket. Sockets created with `socket` function are identified only by their address family.
- On success, returns 0. On error, returns -1 is returned and `errno` is set appropriately.

bind contd.

```
int  bind(int  sockfd, struct sockaddr *my_addr,
   socklen_t addrlen);
```

- Servers usually bind a well-known port, while clients leave the kernel to choose an ephemeral port.
- A process can bind a specific IP address (must belong to an interface of the host):
  - For a TCP client specifies the IP source for packets sent from that socket
  - For a TCP server it restricts the socket to receive incoming client connection destined only to that IP address
- It is normally necessary to assign a local address using `bind` before a SOCK_STREAM socket may receive connections.

getsocketname

```
#include <sys/socket.h>
int getsockname(int socket, struct sockaddr
            *address, socklen_t *address_len);
```

- Retrieves the locally-bound name of the specified socket, stores it in the `address` argument, and its length in the `address_len` argument.
- Truncates the stored address if the actual length of the address is greater than the length of the supplied `sockaddr` structure).
- If the socket has not been bound to a local name, the value stored in `address` is unspecified.
- On success returns 0. On failure returns –1 and `errno` is set to indicate the error.

getpeername

```
include <sys/socket.h>
int getpeername(int socket, struct sockaddr
  *address, socklen_t *address_len);
```

- Retrieves the peer address of the specified socket, stores it in the `address` argument, and its length in the `address_len` argument. Truncation happens if the actual length of the address is greater than the length of the supplied **sockaddr** structure.
- If the protocol permits connections by unbound clients, and the peer is not bound, then `address` will be unspecified.
- On success returns 0. On error returns -1 and sets `errno` to indicate the error.

listen

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

- Called only by a TCP server:
  - Converts un unconnected socket into a passive socket, for which the operating system will accept connections.
  - Specifies the maximum number of connections (`backlog`) that the operating system should queue for this socket. If `backlog` exceeds the implementation-dependent maximum queue length, the length of the socket's listen queue will be set to the maximum supported value.
- On success, 0 is returned. On error, -1 is returned, and `errno` is set appropriately.

accept

```
#include <sys/socket.s>
int accept (int socket, struct sockaddr *address,
   socklen_t *address_len);
```

- Extracts the first connection on the queue of pending connections, creates a new socket with the same socket type protocol and address family as the specified socket, and allocates a new file descriptor for that socket.
- If `address` is not a null pointer, the address of the peer for the accepted connection is stored in `address`, and its length is stored in `address_len`.
- If the protocol permits connections by unbound clients, and the peer is not bound, then the value stored in `address` is unspecified.

accept contd.

```
int accept (int socket, struct sockaddr *address,
   socklen_t *address_len);
```

- Accept can block or not, depending on how the operation is defined for the file descriptor for the socket (synchronously/asynchronously).
- The accepted socket cannot itself accept more connections. The original socket remains open and can accept more connections.
- On success returns the nonnegative file descriptor of the accepted socket. On error returns -1 and `errno` is set to indicate the error.
- When a connection is available, `select` will indicate that the file descriptor for the socket is ready for reading.

select

```
include <sys/time.h>
int select(int nfds, fd_set *readfds,
           fd_set *writefds, fd_set *errorfds,
           struct timeval *timeout);
```

- Indicates which of the specified file descriptors is ready for reading, ready for writing, or has an error condition pending.
- If the specified condition is false for all of the specified file descriptors, `select` blocks, up to the specified `timeout` interval, until the specified condition is true for at least one of the specified file descriptors.
- `nfds`: the range (0 to `nfsd`-1) of file descriptors to be tested.

select contd.

```
int select(int nfds, fd_set *readfds,
           fd_set *writefds, fd_set *errorfds,
           struct timeval *timeout);
```

- `readfds`: if not null, on input specifies the file descriptors to be checked for being ready to read, on output indicates which ones are ready to read.
- `writefds`: if not null, on input specifies the file descriptors to be checked for being ready to write, and on output indicates which ones are ready to write.
- `errorfds`: if not null, on input specifies the file descriptors to be checked for error conditions pending, and on output indicates which ones have error conditions pending.

select contd.

- If `timeout` is not a null pointer, it specifies a maximum interval to wait for the selection to complete. If the `timeout` argument points to an object of whose members are 0, `select` does not block. If `timeout` is a null pointer, `select` blocks until an event causes one of the masks to be returned with a valid (non-zero) value.
- If the time limit expires before any event occurs that would cause one of the masks to be set to a non-zero value, `select` completes successfully and returns 0.
- On successful completion, `select` returns the total number of bits set in the bit masks. Otherwise, -1 is returned, and `errno` is set to indicate the error.

select contd.

```
void FD_CLR(int fd, fd_set *fdset);
```

- Clears the bit for the file descriptor `fd` in the file descriptor set `fdset`.

```
int FD_ISSET(int fd, fd_set *fdset);
```

- Returns a non-zero value if the bit for the file descriptor `fd` is set in the file descriptor set pointed to by `fdset`, and 0 otherwise.

```
void FD_SET(int fd, fd_set *fdset);
```

- Sets the bit for the file descriptor `fd` in the file descriptor set `fdset`.

```
void FD_ZERO(fd_set *fdset);
```

- Initializes the file descriptor set `fdset` to have zero bits for all file descriptors.

select contd.

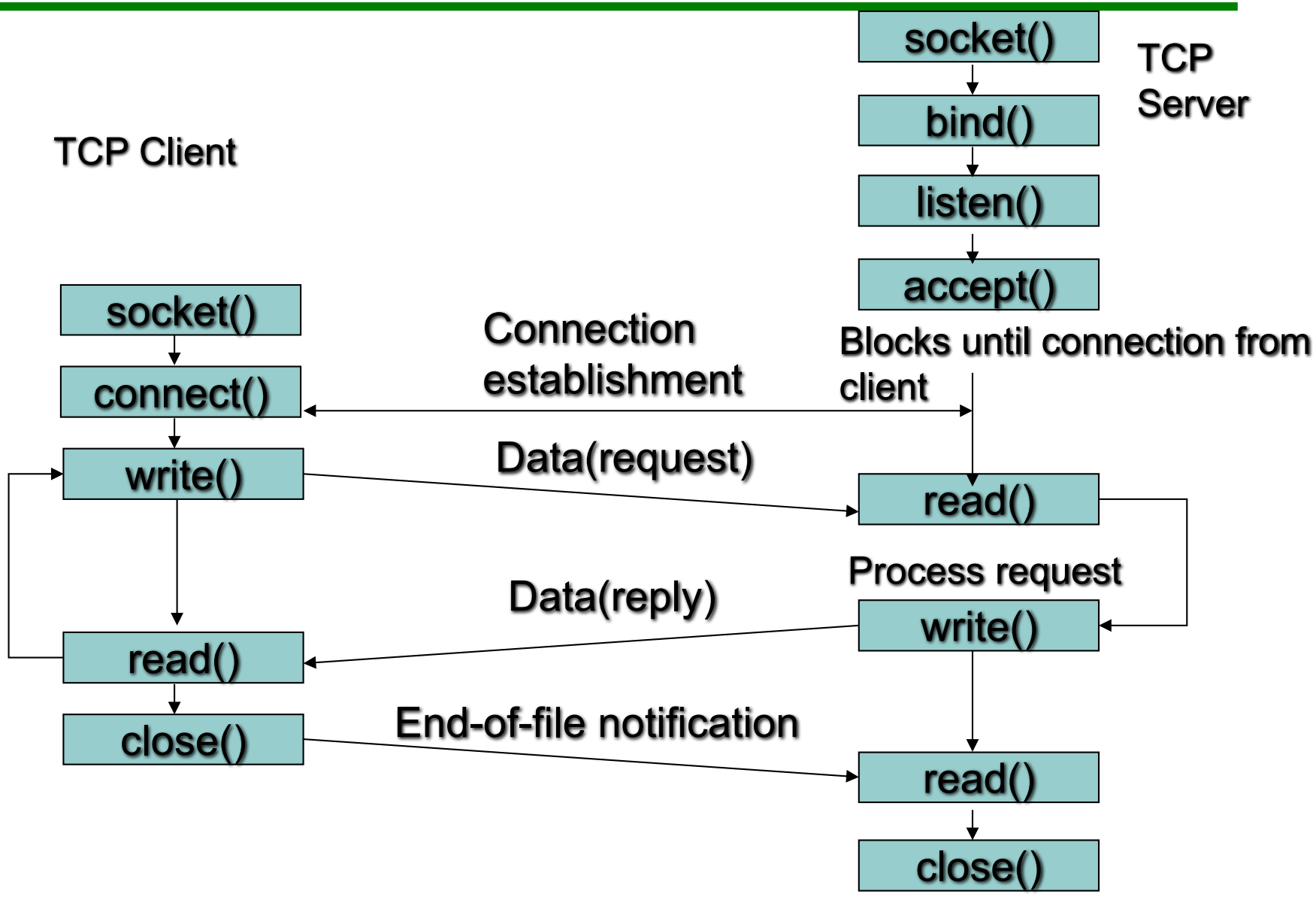
- A file descriptor for a socket that is listening for connections will indicate that it is ready for reading, when connections are available.
- A file descriptor for a socket that is connecting asynchronously will indicate that it is ready for writing, when a connection has been established.

close

```
#include <unistd.h>
int close(int fd);
```

- If `fd` refers to a socket, `close` destroys the socket. If the socket is connection-mode, and the SO_LINGER option is set for the socket with non-zero linger time, and the socket has un-transmitted data, then `close` will block for up to the current linger interval until all data is transmitted.

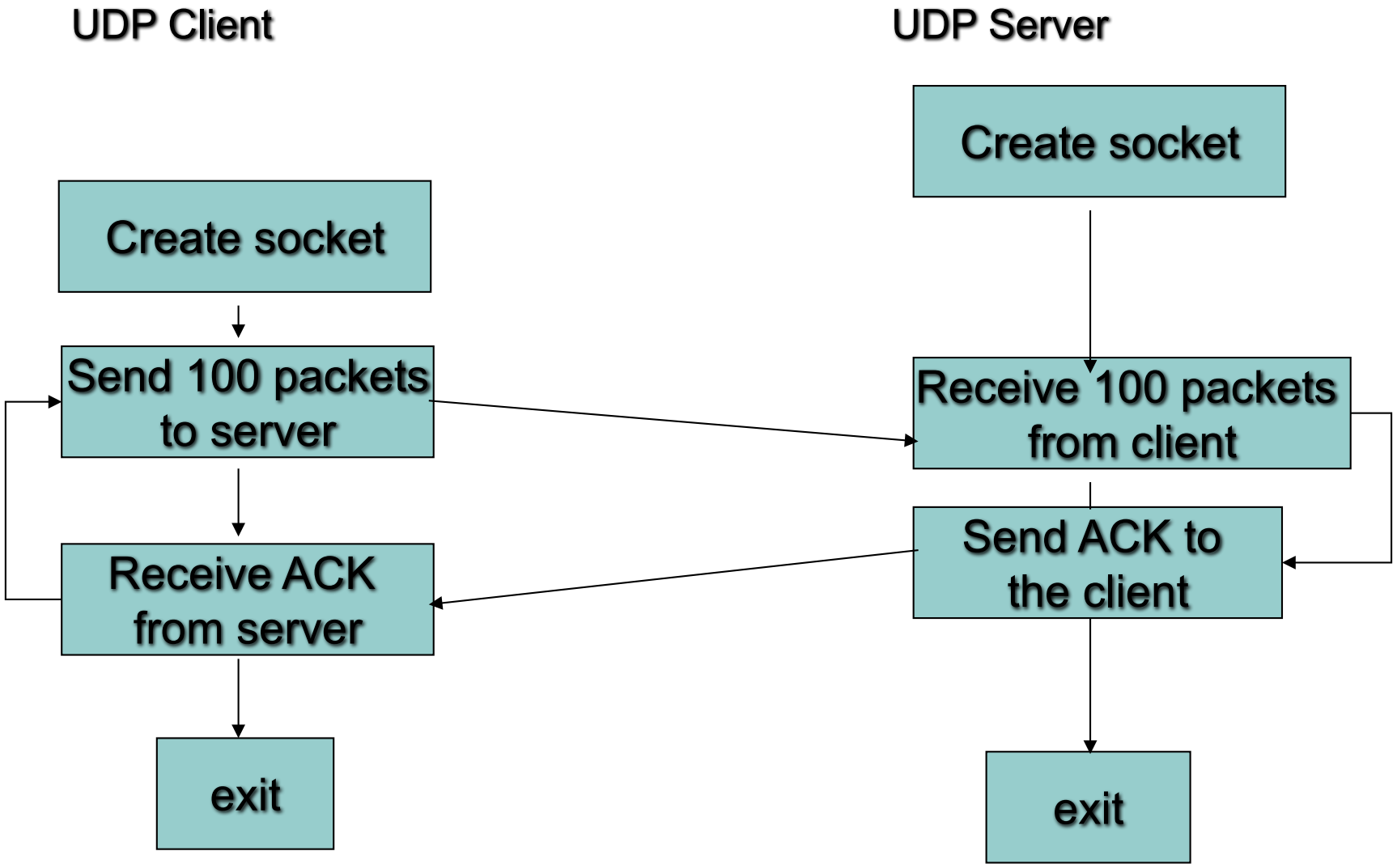
Simple Client-Server TCP App.



UPD Example

- The server
  - Binds to a well-known port
  - Waits for 100 messages from the client
  - Sends an acknowledgement message to the client
  - Exits
- Client:
  - Sends 100 packets to a the server
  - Waits for a an acknowledgement message from the server
  - Exits

Simple Client-Server UDP Example



Revisiting connect

```
#include <sys/socket.h>
int connect(int s, const struct sockaddr *address,
   socklen_t address_len);
```

- Requests a connection to be made on a socket .
  - `s`: the file descriptor associated with the socket.
  - `address`: contains the peer address.
  - `address_len`: the length of the `sockaddr` structure pointed to by the `address` argument.
- On success, returns 0. On failure, returns -1 and `errno` is set to indicate the error.
- If the socket has not already been bound to a local address, connect will bind it: choose and ephemeral port and the IP address if needed.

Revisiting connect contd.

```
int connect(int s, const struct sockaddr *address,
   socklen_t address_len);
```

- UDP (SOCK_DGRAM sockets):
  - `connect` sets the socket's peer address, but no connection is made. The peer address identifies where all packets are sent on subsequent `send`, and limits the remote sender for subsequent `recv` calls.
- TCP (SOCK_STREAM):
  - `connect` attempts to establish a connection to the address `address`.
- If `connect` fails, the state of the socket is unspecified. Close the file descriptor and create a new socket before attempting to reconnect.

Revisiting bind

```
#include <sys/types.h>
#include <sys/socket.h>
int  bind(int  sockfd, struct sockaddr *my_address,
   socklen_t addrlen);
```

- Assigns an address to an unnamed socket. Sockets created with `socket` function are identified only by their address family.
- On success, returns 0. On error, returns -1 is returned and `errno` is set appropriately.

Revisiting bind contd.

```
int  bind(int  sockfd, struct sockaddr *my_addr,
   socklen_t addrlen);
```

- Servers usually bind a well-known port, while clients leave the kernel to choose an ephemeral port.
- A process can bind a specific IP address (must belong to an interface of the host):
  - For a TCP client specifies the IP source for packets sent from that socket
  - For a TCP server it restricts the socket to receive incoming client connection destined only to that IP address
- It is normally necessary to assign a local address using `bind` before a SOCK_STREAM socket may receive connections.

Sending Data

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int s, const void *msg, size_t len, int flags);
int sendto(int s, const void *msg, size_t len, int flags,
                      const struct sockaddr *to, socklen_t
   tolen);
int sendmsg(int s, const struct msghdr *msg, int flags);
```

- Connection-oriented:
  - send, sendto, sendmsg
- Connectionless:
  - sendto, sendmsg

Sending Data contd

- When a message is sent, first it is stored locally by the operating system. If not enough space is available at the sending socket to hold the message to be transmitted:
  - If the sending socket was placed in blocking I/O (socket file descriptor does not have `O_NONBLOCK` set), the sending function blocks until space is available.
  - If the sending socket was place in non-blocking I/O mode (socket file descriptor does have `O_NONBLOCK` set) the sending function fails (`errno` set to `EAGAIN`). The `select` call can be used to determine when more data can be sent.

Sending Data contd.

- All sending functions have similar behavior with respect of returning values:
  - On success they return the number of characters sent.
  - On error they return -1 and set `errno` appropriately (check info pages for details of error codes).
  - Successful completion does not guarantee delivery of the message. A return value of -1 indicates only locally-detected errors.

sendto

```
#include <sys/types.h>
#include <sys/socket.h>
int  sendto(int s, const void *msg, size_t len,
            int flags, const struct sockaddr *to,
            socklen_t tolen);
```

- Sends a message through a connection oriented or connectionless socket (`s`):
  - connectionless socket, the message is sent to the address `to`.
  - connection oriented socket, `to` and `tolen` are ignored.

sendto contd.

```
int  sendto(int s, const void *msg, size_t len,
   int flags, const struct sockaddr *to, socklen_t
   tolen);
```

- Arguments:
  - `s:` the sending socket
  - `to/tolen`: address/length of the target.
  - `msg/len`: messages and length of the message. If the message is too long to pass through the underlying protocol, `sendto` fails and no data is transmitted.
  - `flags:` used to influence the behavior of the function invocation beyond the options specified for the associated socket.

send

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int s, const void *msg,
  size_t len, int flags);
```

- Initiates transmission of a message from the specified socket to its peer.
- Sends a message only when the socket is connected.

send contd.

```
int send(int s, const void *msg, size_t len,
             int flags);
```

- Arguments:
  - `s:` the sending socket
  - `msg/len`: message and length of the message. If the message is too long to pass through the underlying protocol, `sendto` fails and no data is transmitted.
  - `flags:` used to influence the behavior of the function invocation beyond the options specified for the associated socket.

Sendmsg

```
#include <sys/types.h>
#include <sys/socket.h>
int sendmsg(int s, const struct msghdr *msg, int
   flags);
```

- It sends a message through a connection oriented or connectionless socket:
  - If `s` is a connectionless socket, the message is sent to the address specified by `msghdr`.
  - If `s` is a connection-oriented socket, the destination address in `msghdr` is ignored.
- If the message is too long to pass through the underlying protocol, the functions fails and no data is transmitted, `errno` is set appropriately.

Sendmsg contd

- `Msghdr` structure:
  - scatter/gather array structure, avoids additional memory copying.
  - send control information using the `msg_control` and `msg_controllen` members.

```
struct msghdr {
  void    * msg_name;       /*optional address*/
  socklen_t msg_namelen;    /*size of address*/
  struct iovec * msg_iov;   /*scatter/gather array*/
  size_t    msg_iovlen;     /*# elements in msg_iov*/
  void  *   msg_control;    /*ancillary data, see below*/
  socklen_t msg_controllen; /*ancillary data buffer len*/
  int       msg_flags;      /*flags on received message*/
};
```

Sendmsg contd.

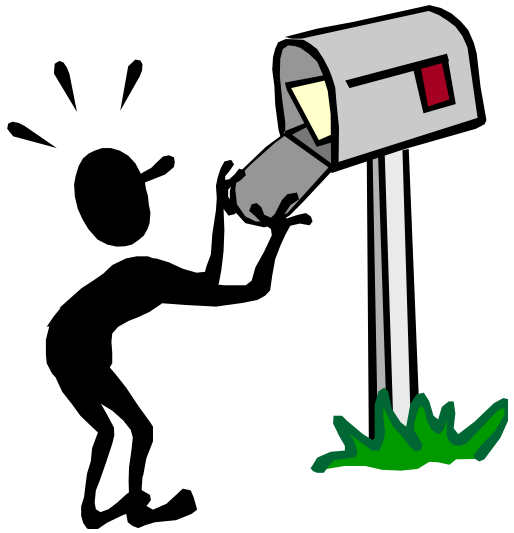
```
int sendmsg(int s, const struct msghdr *msg, int
   flags);
```

- Arguments:
  - `s:` the sending socket
  - `msg`: Points to a `msghdr` structure which contains both a pointer to the buffer containing the destination address, and the address of the buffers which contains the message to be transmitted. The `msg_flags` field is ignored.
  - `flags:` used to influence the behavior of the function invocation beyond the options specified for the associated socket.

Receiving Data

```
#include <sys/types.h>
#include <sys/socket.h>
int recv(int s, void *buf, size_t len, int flags);
int recvfrom(int  s,  void  *buf,  size_t len, int
   flags, struct sockaddr *from, socklen_t
   *fromlen);
int recvmsg(int s, struct msghdr *msg, int flags);
```

- Connection-oriented:
  - recvfrom, recvmsg, recv
- Connectionless:
  - recvfrom, recvmsg



Receiving Data contd.

- All three routines have similar behavior with respect to return values:
  - On success, they return the length of the message.
  - On failure, they return a value of -1 and sets `errno` appropriately.
  - If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from.

Receiving Data contd.

- Blocking/non-blocking behavior if no data is available:
  - If the socket is in non-blocking mode, the call returns -1 and `errno` is set to EAGAIN. The `select` or `poll` call may be used to determine when more data arrives.
  - If the socket is in blocking mode, the receive calls wait for a message to arrive.
- The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested.

Recvfrom

```
#include <sys/types.h>
#include <sys/socket.h>
int  recvfrom(int  s,  void  *buf,  size_t len,
   int flags, struct sockaddr *from, socklen_t
   *fromlen);
```

- Receives a message from a socket and the address from which the data was sent:
  - SOCK_STREAM: as much information is returned as is currently available, up to the size of the specified buffer.
  - SOCK_DGRAM: data is extracted from the first queued message, up to the size of the specified buffer. If the message is larger than the specified buffer, as much data are stored in the buffer as possible and `recvfrom` generates an error message (`EMSGSIZE`). The remainder (or excess) of the message is lost if passed via an unreliable protocol.

Recvfrom contd.

```
int  recvfrom(int  s,  void  *buf,  size_t len,
   int flags, struct sockaddr *from, socklen_t
   *fromlen);
```

- If `from` is nonzero and the socket is not connection-oriented, the network address of the peer that sent the data is copied to `from` and `fromlen` (initialized to the size of from) is modified on return to indicate the actual size of the address stored there.
- If the socket is connection-oriented and the remote side has shut down the connection gracefully, a `recvfrom` call will complete immediately with 0 bytes received. If the connection has been reset, `recvfrom` fails and `errno` is set to ECONNRESET.

Recvfrom contd.

```
int  recvfrom(int  s,  void  *buf,  size_t len,
   int flags, struct sockaddr *from, socklen_t
   *fromlen);
```

- Arguments:
  - `s` : the receiving socket descriptor.
  - `buf/len`: the buffer/length in which to place the message.
  - `flags`: allows the caller to control the reception.
  - `from/fromlen` : socket address structure/length to record the address of the message sender.

Recvmsg

```
#include <sys/types.h>
#include <sys/socket.h>
int recvmsg(int s, struct msghdr *msg, int flags);
```

- Receives a message from a socket and capture the address from which the data was sent.
  - SOCK_STREAM socket: as much information is returned as is currently available, up to the size of the specified buffer.
  - SOCK_DGRAM sockets: data is extracted from the first queued message, up to the size of the specified buffer. If the message is larger than the specified buffer, as much data are stored in the buffer as possible and `recvmsg` generates an error message (`EMSGSIZE`). The remainder (or excess) of the message is lost if passed via an unreliable protocol.

Recvmsg contd

```
int recvmsg(int s, struct msghdr *msg, int
   flags);
```

- If the socket is not connection-oriented, the network address of the peer that sent the data is copied to the `msg_name` and its length is returned in `msg_namelen`; (both are part of the `msg msghdr` structure)
- If the socket is connection-oriented and the remote side has shut down the connection gracefully, a `recvmsg` call completes immediately with 0 bytes received. If the connection has been reset, `recvmsg` fails and returns the error message `ECONNRESET`.

Recvmsg contd.

```
int recvmsg(int s, struct msghdr *msg, int
   flags);
```

- Arguments:
  - `s:` the sending socket
  - `msg`: points to a msghdr structure which contains both the buffer which is to contain the source address, plus the address of buffers into which the incoming message is stored. The `msg_flags` field is ignored.
  - `flags:` used to influence the behavior of the function invocation beyond the options specified for the associated socket.

Recv

```
#include <sys/types.h>
#include <sys/socket.h>
int recv(int s, void *buf, size_t len, int
   flags);
```

- Receives a message from a socket, normally used on a connection mode socket
- Returns as much available information as the size of the buffer supplied can hold.
- If the socket is connection-oriented and the remote side has shut down the connection gracefully, a `recv` call completes with 0 bytes received. If the connection has been reset, `recv` fails and sets `errno` to `ECONNRESET`.

Recv contd.

```
int recv(int s, void *buf, size_t len,
  int flags);
```

- Arguments:
  - `s`: the receiving socket descriptor
  - `buf/len`: buffer/length of buffer where the message will be stored.
  - `flags`: allows the caller to control the reception.

Which Recv/Send to Use?

- Depends on your application: using TCP (connection oriented) or UDP(connectionless)
- Using `msghdr` optimizes the time spend copying buffers of data, however:
  - Portability: `Msghdr` structure not defined on all platforms so for portability reasons your code needs to handle cases when `msghdr` is not defined.
  - Security: More complex when combined with block cipher encryption/decryption . Also the cost of adding such services is much bigger than the optimization obtained by avoiding multiple copying.

Simple Client-Server UDP App.