**Module** **4.1 required reading material:**

* [1] Brian “Beej Jorgensen” Hall, “Beej's Guide to Network Programming, v3.1.11”. April 2023. <https://beej.us/guide/bgnet/html/split/>
  + Read chapter 6.3
* [2] Lewis Van Winkle, “Hands-On Network Programming with C". Packt Publishing. May 2019. ISBN: 9781789349863. <https://learning.oreilly.com/library/view/hands-on-network-programming/9781789349863/>
  + Read chapter 4.
* [3] W. Richard Stevens, Bill Fenner, Andrew M. Rudoff, “The Sockets Networking API: UNIX® Network Programming Volume 1, Third Edition”. Addison Wesley. November 2003. ISBN: 0-13-141155-1. <https://learning.oreilly.com/library/view/the-sockets-networking/0131411551/>
  + This book provides a more in-depth/technical explanation for the topics covered in this module.
  + Read chapter 8. Sections

In this module we’ll cover UDP sockets. Having covered both the socket API in depth and TCP client/server programming, we are now ready to learn about UDP sockets.

**UDP Sockets**

You will notice that there isn’t a lot of new information as far as what functions to use except for two new functions, sendto() and recvfrom(). Most of the new topics we learn from this point forward will draw from the material covered in the two previous modules. That’s why it is important we master those topics. UDP and TCP network programming follow a similar pattern, but there are differences as well.

Both protocols use most of the same socket functions and structures, but UDP uses sendto() instead of send() and recvfrom() instead of recv(). There is no need to call connect() since there is no concept of connection-oriented, we call sendto() and the API sends the data to the recipient.

* UDP and TCP sockets are very similar, having learned about TCP sockets we should be able to write a UDP application after covering a few fundamental differences.
* UDP is a connectionless, unreliable, datagram protocol while TCP is a connection-oriented, reliable byte stream.
* There is no 3-way handshake with UDP.
* There is no connect + send, instead we get sendto(). Although we can use connect() + send() it is important to note they behave differently when using UDP sockets.
* All connect() does with a UDP socket is associated with a remote address. There is no handshake or session establishment.

A diagram of a server

Description automatically generated

**UDP socket functions**

UDP uses recvfrom() and sendto(), these two functions are similar to send() and recv() with the addition of three additional arguments (covered below).

**The recvfrom() function**

The recvfrom() call is used to receive messages from a socket and may be used to receive data on a socket whether it is connection oriented.

Function declaration: ssize\_t recvfrom(int sockfd, void \*buf, size\_t len, int flags, struct sockaddr \*src\_addr, socklen\_t \*addrlen)s

* Parameter
  + sockfd: file descriptor returned from socket().
  + \*buf: A pointer to the buffer where the received data will be stored
  + len: The maximum length of the data that can be received and stored in buf.
  + flags: Optional flags that can modify the behavior of the recvfrom() function.
  + \*src\_addr: A pointer to a struct sockaddr that will be filled with the source address of the sender. In connectionless protocols like UDP, this is important for knowing where the data came from.
    - In short, this structure contains the protocol address (IP address and port number) of the system connecting/sending the data to this socket. This structure tells who sent the datagram or who initiated the connection.
* Returns
  + Returns the length of the message on successful completion. 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.
  + The number of bytes received, or -1 if an error occurred. If the socket is in non-blocking mode and no data is available, recvfrom() will return -1 with errno set to EAGAIN or EWOULDBLOCK.
  + Writing a datagram of length 0 is acceptable. This means that a return value of 0 from recvfrom() is acceptable and does not mean the peer has closed the connection, as does the return value of 0 from recv(). [source: Unix Network Programming + man pages]

Notes

recvfrom() places the received message into the buffer buf. The caller must specify the size of the buffer in len. If src\_addr is not NULL, and the underlying protocol provides the source address of the message, that source address is placed in the buffer pointed to by src\_addr. In this case, addrlen is a value‐result argument. Before the call, it should be initialized to the size of the buffer associated with src\_addr. Upon return, addrlen is updated to contain the actual size of the source address. The returned address is truncated if the buffer provided is too small; in this case, addrlen will return a value greater than was supplied to the call.

**The sendto() function**

The sendto() function sends data on the socket with descriptor socket. The sendto() call applies to either connected or unconnected sockets.

Function declaration : ssize\_t sendto(int sockfd, const void \*buf, size\_t len, int flags, const struct sockaddr \*dest\_addr, socklen\_t addrlen)

* Parameters
  + sockfd: file descriptor returned from socket().
  + \*buf: The pointer to the buffer containing the message to transmit.
  + len: The length of the message in the buffer pointed to by the buf parameter.
  + flags: Optional flags that can modify the behavior of the sendto function.
  + \*dest\_addr: A socket structure containing the protocol address of where the data is to be sent.
    - This structured call filled in by a call to recvfrom(), manually by the user or by a call to getaddrinfo().
  + addrlen: The size of the dest\_addr structure.
* Returns
  + On success, these calls return the number of characters sent. On error, -1 is returned, and errno is set appropriately.
  + A value of 0 or greater indicates the number of bytes sent, however, this does not assure that data delivery was complete. A connection can be dropped by a peer socket and a SIGPIPE signal generated at a later time if data delivery is not complete.
  + No indication of failure to deliver is implied in the return value of this call when used with datagram sockets.

Notes:

With a UDP socket, the first time the process calls sendto(), if the socket has not yet had a local port bound to it, that is when an ephemeral port is chosen by the kernel for the socket. [Unix network programming, chapter 8.6, pg.: 294]

**UDP client design**

Two different ways to structure a UDP client.

1. Using connect(), send(), and recv().
   1. In this mode, the client only receives data from the peer having the IP address and the port that is given to connect. This can be useful if we want to limit the hosts we want to communicate with.
2. Using sendto() and recvfrom().
   1. In this mode, recvfrom() returns data from any peer that sends data to the IP address of the interface/system.
   2. Can also use bind() to lock down address.

Both methods require the application calls the socket() function.

The client does not establish a connection with the server, instead the client just sends a datagram to the server using the sendto() function, which requires the address of the destination (server) as a parameter. UDP makes no effort to make there are no errors or that the data makes it to its destination, that is left up to the programmer.

**UDP Client Example** (only the three socket functions mentioned above):

int socket\_a**;**

socket\_a **=** socket**(**family**,** sock\_type**,** protocol**)**

sendto**(**socket\_a**,** data**,** data\_length**,** flag**,** peer\_address**,** peer\_address\_length**);**

recvfrom**(**socket\_a**,** data**,** data\_length**,** flag**,** peer\_address**,** peer\_address\_length**);**

Difference between a TCP client and a UDP Client:

A diagram of a computer

Description automatically generated

**UDP Server**

UDP Server methods:

1. UDP servers only need one socket to communicate with any number of peers.
2. UDP servers don't require listen() and bind() to wait for and establish new connections. These functions are not used.
3. Use bind() to bind the socket to a local address.
4. Use select() to check/wait for data but only need to monitor one socket instead of multiple sockets like in TCP.

A diagram of a server

Description automatically generated

**UDP Server Example** (socket(), bind(), sendto(), recvfrom())

int socket\_a**;**

socket\_a **=** socket**(**family**,** sock\_type**,** protocol**)**

bind**(**socket\_a**,** bind\_address**,** bind\_address\_len**);**

#

sendto**(**socket\_a**,** data**,** data\_length**,** flag**,** peer\_address**,** peer\_address\_length**);**

#

recvfrom**(**socket\_a**,** data**,** data\_length**,** flag**,** peer\_address**,** peer\_address\_length**);**

**How are sockets, addresses + ports bound in UDP?**

sendto() will ask the kernel for this information the first time this function is called. We can also use connect() or bind() if we want to have control of the port number.

Once a port is chosen by the kernel for a given host using sendto(), that port never changes for the duration of that call. That is, as long as we're communicating with that host using that given socket file descriptor.

**Lost Datagrams**

Recall, UDP is not reliable and there is no guarantee that a client or server will send or receive all the data. A typical way to prevent recvfrom() from not receiving all its intended data is to place a timeout on the client's call to recvfrom(), although this is not an absolute solution.

This concludes our coverage of UDP sockets.

**Assignments**

* Rewrite the listener.c program from “Beej's Guide to Network Programming” Chapter 6.3
* Chapter 4 from “Hands-On Network Programming with C", rewrite the programs to work on Linux only.
* Write a UDP client and server program.
* Add select() to the previous program.
* Add the ability to handle sending and receiving partial data.