**Module 2.4 – Introduction to the Socket API part 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/>
  + Chapter 5.
* [2] 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 4 – Elementary TCP sockets.
* [3] Linux man pages. Either run man <function name> or go to either <https://www.die.net/> or <https://man7.org/linux/man-pages/index.html> and search for the function by name.

\*Optional

* [4] Jon Erickson, “Hacking the Art of Exploitation 2nd ed”. No Starch Press. February 2008. ISBN: 978-1593271442. <https://learning.oreilly.com/library/view/hacking-the-art/9781593271442/>
  + Read chapter 0x04, section 0x420 -Sockets.
  + Provides basic information but not as in-depth as the first two books.

In this module we dig deeper into the main socket API functions by covering each of them and fully explaining each parameter, return type, and cover any gotchas or technical details. In the previous module we covered the main socket structures we'll be using throughout the course. We're now ready to start learning about the different socket functions and start writing simple socket applications.

**Goals:**

In this section we'll learn about the elementary socket functions required to write a complete TCP client and server.

***This module will require a lot of reading and research and will likely take some time to get through. Mastering these concepts will prepare us for the rest of the modules in the course since any application we write will make use of all the functions covered here. As we progress through the later modules, our focus will be more on the practical side of things and less on the theory.***

**Server-Client paradigm**

For the most part, TPC/UDP applications written in the Server-Client paradigm will follow the set of steps presented in the images below. We can use these two diagrams as an aid to remember how to structure our socket application following that model. We can see how the previously covered socket APIs are called and used through the application life cycle.

A diagram of a computer

Description automatically generatedA diagram of a computer process

Description automatically generatedFlow of a UDP server/client program Flow of TCP server/client program:

We'll refer to the TCP and UDP Program flow images throughout the rest of this section and I recommend using it as you progress through each subsequent module.

The general flow of a TCP a server, follows these set of steps:

1. Declare and initialize the relevant socket structures (covered in the previous module) using getaddrinfo() or similar.

2. Instantiate a socket file descriptor using the socket() function.

3. Bind our socket to an interface or all interfaces using the bind() function .

4. Put the socket in a listening state using the listen() function.

5. Accept remote connections on the given socket (address and port pair) using the accept() function.

6. Receive and/or send information (depending on the server's functionality) using the recv() and send() functions, respectively.

7. Finally, teardown/close the connection using the close() function.

**Socket Functions**

[*The reading materials do a great job at explaining each of the socket functions, recommend reading those. We'll make note of each function here and try to add information we thought was lacking or try to clarify sticking points.*]

**The socket() function:**

To perform network I/O, the first thing a process must do is call the socket function to specify the type of communication (protocol) desired. The socket function provides this functionality.

**Function declaration:** socket(int domain, int type, int protocol):

* Purpose:
  + Used to create and initialize a new socket (communication channel between endpoints), returns a file descriptor for the socket or -1 on error.
  + \*This function creates the socket binding interface and returns the FD assigned to it.\*
* Parameters: The parameters for this function allow you to specify what kind of socket you want to use, protocol, etc.
  + domain (you might see family instead in older literature or legacy code):
    - This parameter specifies the address family, which determines the type of addresses the socket can communicate with. IPv4 vs IPv6, for example.
    - Possible values:
    - It can be one of the following:
    - AF\_INET: for IPv4 protocols
    - AF\_INET6: for IPv6 protocols.
    - AF\_UNSPEC: allows the system to choose the appropriate family based on available resources.
    - AF\_UNIX: Unix sockets for local communication.
* type: The type of socket to use for the connection/socket.
  + SOCK\_STREAM for TCP.
  + SOCK\_DGRAM for UDP.
  + SOCK\_RAW for raw sockets.
  + Others, as supported.
* protocol: The protocol to use.
  + It can be TCP, UDP, or other.
  + Although it is generally set to zero (0) to let the system choose the default protocol for the given address family and socket type.

Returns: On success, returns a non-negative integer value for the socket file descriptor or -1 on error.

Notes:

* The domain parameter is tightly coupled with each protocol because each domain (protocol family), supports its own set of protocols.
* For example, AF\_INET can support both TCP and UDP and will use either SOCK\_STREAM or SOCK\_DGRAM.
* AF\_XXX vs PF\_XXX: Undoubtedly, you will see either AF\_INET or PF\_INET used for the domain argument used in the socket() function. so here is the explanation behind it:
  + "The "AF\_" prefix stands for "address family" and the "PF\_" prefix stands for "protocol family." Historically, the intent was that a single protocol family might support multiple address families and that the PF\_ value was used to create the socket and the AF\_ value was used in socket address structures. But in actuality, a protocol family supporting multiple address families has never been supported and the <sys/socket.h> header defines the PF\_ value for a given protocol to be equal to the AF\_ value for that protocol. While there is no guarantee that this equality between the two will always be true, should anyone change this for existing protocols, lots of existing code would break." [Unix Network Programming - 4.2, pg. 133]
* Older code will hardcode the arguments to socket while modern code will use a struct addrinfo’s ai\_family, ai\_sockettype, and ai\_protocol parameters to populate these values.

socket(AF\_INET, SOCK\_STREAM, 0) vs socket(res->ai\_family, res->ai\_socktype, res->ai\_protocol), where res is an addrinfo structured populated by getaddrinfo().

**The bind() function:**

This function binds a socket so it can listen for incoming connections.

**Function declaration:** int bind(int sockfd, struct sockaddr \*my\_addr, int addrlen):

* Purpose
  + Once a socket has been created with socket(), we use bind() to associate a socket with a port on the local machine. That is, we bind the pair to the interface. This call is usually followed by the listen() function on server applications and rarely used on client applications.
* Parameters
  + sockfd: socket file descriptor returned by socket().
  + struct sockaddr \*my\_addr: a pointer to a protocol-specific address. This is a structured pre-populated by a call to getaddrinfo().
  + addrlen: the size of the sockaddr structure.
* Returns
  + 0 on success or -1 on failure.
* Notes:
  + We can specify the port to bind the socket to by supplying the number to the "service" parameter for getaddrinfo().
  + On a system with multiple addresses, we can tell bind() which address to bind to by specifying the address in the "node" parameter for getaddrinfo().
  + Bind is usually called by server programs and not by clients. If bind() is not invoked, the kernel will choose an ephemeral port for the client application.
    - For a TCP client, bind will assign the source IP address that will be used to deliver the IP datagrams.
    - For a TCP server, we can restrict the socket to listen or accept connections on a specific interface/address.
    - For TCP clients, we can use bind() to specify which interface our packets should be sent out of. In the infosec world this is useful in networks where network traffic to certain devices might be limited to specific source networks (ports or addresses).
      * Picture a system with multiple interfaces/addresses, such as 192.168.1.2 and 172.16.2.2 for interfaces A and B, respectively. If we are trying to reach a network that requires that we come from the 172.x address, we can use bind to make sure our traffic is sourced from this network. Traffic tunneling applications such as network use bind() in this fashion.
    - For server applications, this port is used by the kernel to track and match incoming packets to a process' socket descriptor. This is important because a server application can handle multiple client connections on the same port.

**The connect() function:**

The connect function is used by a TCP client to establish a connection with a TCP server.

**Function declaration:**  int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen).

* Parameters: The parameters are the same type we've used in the previous functions; the only difference is that the sockaddr \*addr parameter must contain the IP address and port number of the remote host.
  + sockfd: socket file descriptor returned by socket().
  + struct sockaddr \*addr: a pointer to a protocol-\*\*specific\*\* address. This is a structured pre-populated by a call to getaddrinfo().
  + addrlen: the size of the sockaddr structure.
* Returns
  + The function returns only when the connection is established, or an error occurs. Returns 0 on binding or connection success, -1 on failure. errno is set to indicate the error number. It's good practice to check errno.

**The listen() function:**

listen for connections on a socket.

Function declaration: int listen(int sockfd, int backlog).

* Purpose:
  + Used by server applications or applications listening for incoming connections.
  + Usually called after the socket() and bind() functions and must be called before accept().
  + This function performs two actions (see active vs passive sockets below):
  + Puts the socket in a state where it listens for new connections (from closed to listen).
  + t specifies the maximum number of connections the kernel should queue for this socket (see backlog below).
* Parameters:
  + sockfd: is the usual socket file descriptor returned by socket().
  + backlog: is the number of connections allowed on the incoming queue.
* Returns:
  + On success, zero is returned. On error, -1 is returned, and errno is set to indicate the error.

Notes:

Active vs Passive sockets:

When a socket is created with socket(), it is assumed to be an active socket (the application will issue a connect). listen() converts the socket from active to passive.

* Active: known as a client socket, typically used to initiate a connection to a remote server.
* Passive: Known as a server socket, used to wait for connections.

backlog (see 4.4 from Unix Network Programming):

* This is the number of connections that will be put in a queue until accept() sees them and acts on them.
* Two queues exist:
  + Incomplete connection: contains an entry for each SYN packet that has arrived from a client.
  + Completed connection contains an entry for each client with whom the TCP three-way handshake has completed (established connection).

**TCP Listening Socket Queues Diagram:**

A diagram of a network

Description automatically generated

**The accept() function:**

accept() is called by a TCP server to return the next completed connection from the front of the completed connection queue. (Completes a connection from a remote client.)

Function declaration: int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen).

* Parameters:
  + sockfd: the usual socket file descriptor returned by socket().
  + struct sockaddr \*addr: used to return the protocol address of the connected peer (address + port).
    - A struct sockaddr\_storage is more commonly used, we cast it to (struct sockaddr \*) pointer to sockaddr.
  + addrlen: the value indicating the size of the returned structure. (size of struct \*addr)
* Returns:
  + Returns 0 on success or -1 on failure and sets errno to the error number.
  + Returns up to three values:
    - The protocol address of the client process.
    - An integer value that is either a new socket or an error indication.
    - The size of this address if the structure is not big enough to hold the returned address.

Notes:

* By default, accept() blocks until a new connection is made (the program will sleep until a connection is made to the listening socket).
* When a connection is made it creates a new socket for it.
  + The New file descriptor is created by the kernel.
* The listening socket will remain in the listening state and the new socket will be used to handle the received connection for the given client.
  + It creates a socket for each connection that is accepted.
* The original socket used for the calls to bind() and listen() is called the listening socket while the new socket returned by accept() is called the connected socket.
* This new socket can be used to send and receive data.
* accept() fills the sockaddr \*addr structure with the remote system's address and port.
* The socket is closed when the server is finished serving a given client.

**The send() function**

Sends a message (data) on a socket.

Function declaration: int send(int sockfd, const void \*msg, int len, int flags).

* Purpose
  + This function takes the client's socket, a pointer to the data to be sent, and the length of the data to send.
* Parameters
  + sockfd is the usual socket file descriptor returned by socket().
  + \*Msg : is a pointer to the data you want to send.
  + len : is the length of the data in bytes.
  + flag: flags for I/O functions, can usually be set to 0. See man pages for more details.
* Returns
  + The number of bytes actually sent out (might be less than the number it was told to send). Or -1 on error, and errno is set.

**The recv() function**

Receives a message (data) from a socket.

Function declaration: int recv(int sockfd, void \*buf, int len, int flags)

* Purpose
  + This function is similar to send() but it receives data instead of sending it.
* Parameters
  + Sockfd: is the usual socket file descriptor returned by socket().
  + \*buf : is a pointer to the buffer to read the data into (write to).
  + len : is the length of the data in bytes.
  + flag: flags for I/O functions, can usually be set to 0. See man pages for more details.
* Returns
  + The number of bytes received.
  + 0 if the remote side has closed the connection.
  + Or -1 on error, sets errno.
* Notes
  + Data is read from sockfd into \*buf, len is the maximum length of the buffer.

**The close() function**

This is the normal Unix close function, and it is used to close a socket and terminate a TCP connection.

Function declaration: close(sockfd)

* Purpose
  + Closes a file descriptor.
* Parameter
  + sockfd is the usual socket file descriptor returned by socket().
* Returns: N/A
* Notes
  + The default behavior is to mark the socket as closed and return to the process immediately.
  + Makes the socket descriptor no longer usable by the process. It cannot be used with any of the I/O operation functions.
  + TCP will still try to send data queued to be sent to the other end and after it completes, the normal TCP connection termination sequence takes place.
  + When a process closes the connected socket, it decreases the reference count for the descriptor. If there are multiple references for a given socket, a single close() call to it won't close it or shut it down and no TCP connection termination will take place.
  + The shutdown() (see below) function will forcibly close the connection.

A diagram of a computer program

Description automatically generated

**The shutdown() function**

shut down socket send and receive operations. The shutdown() function shall cause all or part of a full-duplex connection on the socket associated with the file descriptor socket to be shut down.

Function declaration: int shutdown(int sockfd, int how)

* Purpose:
  + Gives more control over how a socket closes. It allows you to cut off communication in a certain direction or both. It doesn't actually close the file descriptor but rather changes its usability.
* Parameters
  + sockfd: is the usual socket file descriptor returned by socket().
  + how: Specifies the type of shutdown. The values are as follows:
    - SHUT\_RD
      * Disables further receive operations.
    - SHUT\_WR
      * Disables further send operations.
    - SHUT\_RDWR
      * Disables further send and receive operations.

This concludes our overview of the network APIs for a TCP server/client application. We are now ready to start writing TCP and UDP applications.