Sina Communication Systems

Socket Programming in C

Start Date: 02/13/2024

by

Mahdi Ghiasi
mmgh74@gmail.com

Contents

1 Socket Creation

A socket is one endpoint of a two-way communication link between two programs or processes. In other words, it's a way to talk to other programs using standard Unix file descriptors. In Unix, every I/O action is done by writing or reading a file descriptor. A file descriptor is just an integer associated with an open file and it can be a network connection, a text file, a terminal, or something else.

To a programmer, a socket looks and behaves much like a low-level file descriptor. This is because commands such as read() and write() work with sockets in the same way they do with files and pipes.

When creating a socket in the C programming language, we call the "socket" function which gets three input arguments: domain, type, and protocol. We will discuss more details of these three arguments in the following paragraphs.

- **Domain:** A socket is recognized by the other sockets outside its local program from the address associated with it. Therefore, a socket must have an address associated with it to receive data from outside its local program. The "domain" parameter in the "socket" function indicates the type and the structure of the socket's address. Although there are many types of addresses defined in the "socket.h" header file, four of them are used mostly:

    1. **AF_INET:** This domain is used when we want to establish communication between processes over the Internet using IPv4 addressing structure.
    2. **AF_INET6:** This domain is the same as the AF_INET domain except that we want to use the IPv6 addressing structure for processes communicating over the Internet.
    3. **AF_UNIX:** This domain structure is used when processes are in the same host (more precisely, the same Operating System) and want to communicate locally without a network connection. This domain structure uses the OS file system for addressing sockets. This means the address of a socket in this domain structure is a file path in the OS.
    4. **AF_LOCAL:** This domain structure is used when processes want to have network communication in the same host. In other words, processes use the loop-back network interface to exchange data. This domain is a subset of the AF_INET domain. It means there is no difference between using AF_INET domain with "127.0.0.1" IP address and using AF_LOCAL domain.

- **Type:** This parameter determines the type of communication semantics used by the socket. Each type of socket has some specific properties and can support several protocols. A default protocol has been indicated for each type of socket. Many types of sockets have been defined in the "socket.h", but four of them are used regularly:

    1. **SOCK_STREAM:** This type of socket provides two-way, reliable, connection-based, byte-stream transmission. Therefore, you can use protocols that can be supported by the mentioned properties (ex: TCP). The default protocol for this type of socket is TCP [1].
    2. **SOCK_DGRAM:** This type of socket provides unreliable, connectionless datagram transmission. Connectionless protocols that use datagrams for exchanging data (such as UDP) should use this type of socket[2]. The default protocol for this type of socket is UDP[3].
    3. **SOCK_RAW:** A raw socket allows applications to directly access lower-level network protocols such as ICMP. Raw sockets are typically used for transferring network control packets (like ICMP, OSPF, BGP, ...) and implementing custom network protocols. When using a raw socket, the network protocol (TCP, UDP, ...) must be implemented manually. That's why we use this type of socket for exchanging network control packets. With these sockets, we can build our

[1]Apart from TCP, there are other connection-based protocols so we must have a stream socket type for using them.
[2]Connection-oriented protocols are not supported by datagram sockets.
[3]Apart from UDP, there exist other connectionless protocols that use datagrams for data exchange so we must deploy datagram sockets to implement them.

network header according to our desired protocol (ICMP, OSPF, ...). Raw sockets don't support the connection-oriented property but we can implement connection-oriented protocols, like TCP, with them. In this scenario, the developer should implement the handshaking process manually.[4]

4. **SOCK_SEQPAKCET:** This type of socket provides a two-way, connection-based, reliable byte transmission. It's used when the integrity and the order of data is crucial. It's another socket type that supports the TCP protocol and is commonly used with AF_UNIX domains.

- **Protocol:** This parameter specifies the communication protocol to be used by the socket. In other words, it defines the rules and conventions for communication between the endpoints of the socket. Common protocols are discussed below:

  1. **IPPROTO_IP (0):** When you set the protocol parameter to zero or IP, it means you want the default protocol to be used for that type of socket.[5]
  2. **IPPROTO_TCP:** It indicates that the endpoints' sockets want to deploy the TCP transport protocol for communication.
  3. **IPPROTO_UDP:** It indicates that the endpoints' sockets want to deploy the UDP transport protocol for communication.
  4. **IPPROTO_RAW:** It indicates that there's no transport protocol specified for sockets' communication and the developer wants to implement his custom protocol manually.

Here are some points we should consider when creating a socket:

- There are many network control packet types (ICMP, OSPF, BGP, ...) that have their protocol for information exchange. Some of them like ICMP are defined in the "in.h" header file but in most cases, the developer has to construct the control packet structure manually based on the desired protocol.
- You cannot use IPPROTO_RAW protocol for sockets with AF_UNIX domain. That's because RAW protocol (and respectively raw sockets) is used for exchanging control packets and implementing custom communication protocols. There's no network connection in the case of AF_UNIX domains so we don't have control packets, and implementing a custom communication protocol doesn't have any point.
- As said before, an address in the AF_UNIX domain is nothing but a file path in the OS. Although an existing binary file will be associated with the socket as its address, nothing is written in it and it only indicates the socket's address.
- For AF_UNIX domains, the only valid entry for protocol is 0. The OS kernel is responsible for data exchange and the data are sent and received via system calls. Therefore, specifying a communication protocol for sockets with AF_UNIX domain is meaningless. [6]

2 Address Binding

A socket is recognized from outside its local program by the address associated with it[7]. As a result, a socket must have an address associated with it if it wants to receive data from outside its local program. Consequently, we must know the address of a socket if we want to send data to it.

We associate an address with a socket by using the "bind" function that is defined in the "socket.h" header file. Sample codes for binding a socket to an address can be found in (GitHub).

Here are some points we should take into account about address binding:

- It's not necessary to bind an address with a socket if you want to send data.

[4]You must have root or administration access to the OS for creating a raw socket.
[5]Remember that we had a default protocol for each type of socket.
[6]If you set the protocol to ICMP, it will work, but that doesn't mean you're exchanging ICMP packets.
[7]In the case of network connection, a port number is also associated with a socket.

- When you close a socket, it will take a while for the OS to free the associated address. Therefore, it's better not to bind an address to a socket when it's not needed (more precisely, when you just want to send data).
- When you are using a connection-oriented protocol (like TCP), the OS will automatically bind an address to the client (the side that is sending data) after the connection is established (accepted by the server). That's because, in a connection-oriented protocol, both sides have to send and receive data from each other for handshaking. As we said before, a socket cannot receive data when there's no address associated with it. As a result, in a connection-oriented protocol, both sides must have been bound by an address.
- In a connection-oriented protocol (like TCP), if the address of one side of the connection changes, the connection will be terminated and a new connection will be established automatically.
- Before binding an address to an AF_UNIX domain socket, we must unlink the file path we want to use as the socket address. To this mean, we use the "unlink" function in the "unistd.h" header file. This function will clean all the file's contents and will prevent all other processes from writing on it[8].

3 Establishing Connection

In the connection-oriented protocols, we must establish a connection for exchanging data.

1. **TCP:** TCP is the most famous connection-oriented transport protocol used in network connections. We will discuss this protocol in the following paragraphs:

   - **Network view:** In the network layer, a handshaking process will take place between the server and client. First, the client (the side that wants to request a connection) will send a SYN (synchronization) packet to the server indicating that it wants to start a connection. Then, the server will respond with an ACK (acknowledgment) packet alongside another SYN packet. At last, the client will send an ACK packet as a response to the server's SYN packet and the handshaking process ends.
     During data exchange, the sender sends an ACK packet alongside other data and the receiver should respond with another ACK to confirm that it has received the data. The sender will continue to send its data periodically until it receives an ACK packet from the receiver. In other words, the sender will continue sending its data until it makes sure that the receiver has got it. That's why we know the TCP protocol as being reliable.
     To terminate the connection, one side will send a FIN (finish) packet alongside an ACK packet indicating that it wants to end the connection. Then the other side will respond with another ACK and FIN packet and the connection will be terminated.
   - **Application view:** In the application layer, we should listen on the server socket. This is done by calling the "listen()" function. This function will activate a socket and enable it to receive connection requests. This function is non-blocking and it only changes the mode of a socket. It's important to know that we can only call this function on sockets whose type supports connection-oriented protocols. For instance, we can listen on a stream socket but we cannot listen on a datagram or raw socket. You can see the syntax of this function in the code base.(GitHub)
     On the client side, we have to send a connection request to the server. This is done through calling the "connect()" function. The syntax of this function can be found here: (GitHub)
     When we call the listen function in the server, it starts receiving connection requests, and those requests will be entered in a queue. For handling connection requests, we should call the "accept()" function in the server. This function will respond to one connection request in the queue. This is a blocking function and it stops the program until it finds a request in the queue. The syntax of this function can be viewed here: (GitHub). When the server accepts the connection request, the client and server can exchange information.

[8]If the file doesn't exist, it will be created.

4 Receiving and Sending Messages

There are some predefined functions in the "socket2.h" header file for receiving messages by the socket file descriptor.
In the connection-oriented protocols, we must use the "read()" function for this end. We don't need to pass the source address to this function because the socket itself has already been connected to the destination. Therefore, we can receive messages by just knowing the socket ID.
In the connectionless protocols, we must use the "recvfrom()" (receive from) function. In this scenario, we need to know the source address because there is no connection between the socket and the messages' source.
Both of the mentioned functions are blocking. It means they will stop the process until they receive a message from the identified source (We could say, they act like the well-known "scanf()" function).
The syntax of these functions can be seen here (GitHub🔗).

In order to send messages by the socket file descriptor, there are some predefined functions in the "socket2.h" header file.
In the connection-oriented protocols, we must use the "send()" function to this end. We don't need to pass the destination address to this function because the socket itself has already been connected to the destination. Therefore, we can send messages by just knowing the socket ID.
In the connectionless protocols, we must use the "sendto()" function. In this scenario, we need to know the destination address because there is no connection between the socket and the messages' destination.
The syntax of these functions can be seen here (GitHub🔗).

References