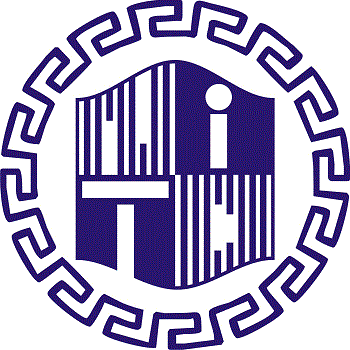
ASSIGNMENT 05

**Computer Science & Engineering Department**

**National Institute of Technology, Delhi**



**NETWORK PROGRAMMING**

**Submitted By: Submitted To: MD SARFARAJ ALAM**

**Dr Ravi Arya Roll NO:171210037**

**(Assistant Professor) Branch CSE**

**NIT DELHI Date: 30/05/2020**

**Write a report of 500 words on Raw Sockets**.

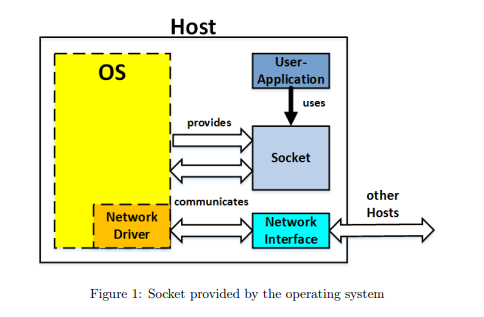
**1 Introduction**

First of all we want to define the basic wordings and concepts used. Then the reader is given a short overview about the possibilities and restrictions the available APIs impose on the user.

Internet sockets are the common way to perform network communication implemented in most operating systems.

They are usually provided by a socket API and are based upon the same principles as reading and writing a file.

This function then returns a socket descriptor, usually a simple integer, similar to the ones provided by most operating systems for Read- and WriteOperations on files. This socket descriptor then can be used to write or read data from the socket.



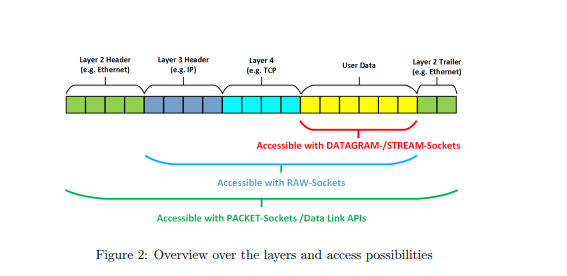
**RAW-sockets**

RAW-sockets are part of the standard Berkeley sockets and the socket API that is based upon it.

They are another option in addition to the already mentioned DATAGRAM- or STREAM-sockets to create data packets with the socket API.

In addition to simply sending data and defining address information the RAW-socket allows the user to access and manipulate the header and trailer information of the lower layers, more specifically with RAW-sockets to the Network and Transport layer (layer 3 and 4 of the OSI model).

Since RAW-sockets are part of the Internet socket API they can only be used to generate and receive IP-Packets.



**The biggest problem with RAW-sockets is that there is no uniform API for using RAW-sockets under different operating systems:**

• The provided APIs differ in regard to the used byte order. Depending on the operating system the fields of the packets have to be filled with data in network- or host-byte-order.

• Differences also exists in the types of packets and protocols that can be created using the RAW-socket API.

• The usage and functionality of the APIs is also different for each operating system.

• Some operating systems do not allow certain packet types to be received using the RAW-socket API.

• There are also differences in the definitions and paths of the necessary header files provided by each operating system.

• The required access levels for using the RAW-socket API can also differ. However most operating systems require root or admin access permissions to use them.

**Implementations for different Operating Systems:**

1. **Windows:**

Windows as a operating system is pretty restricted when it comes to RAW-socket, PACKET-socket and Data Link Layer programming. In general it is difficult to get the programs running under Windows and the available options are pretty limited. For that reason in general 3rd party libraries are recommended if the user wants to do network programming on Windows systems and still write portable code.

1. **Linux:**

Linux is one of the operating systems for which it is easy to do RAW-socket and Data Link Layer Programming. It provides the APIs to do both, but the kernel has to be compiled with the option to support the options.

* 1. **RAW-sockets:**

The RAW-socket is included in the socket-API as we already discussed. It following items still have to be observed:allows both sending and receiving of RAW-packets, however the

• Due to essential nature of header information for networking functionality and security using RAW-sockets requires root access permissions [5]. • The ports of the network layer are not endpoints anymore since RAW-sockets work on the layers below. Filtering based on ports has to be done manually .

• The bind() and connect() functions are no longer necessary . bind() and textttconnect() can still be used to the define source address and target address to be entered automatically by the kernel [5]. Additionally a raw socket can be bound to a network device using SO\_BINDTODEVICE.

• The listen() and accept() functions are without function, since the client-Server-Semantic is no longer present [5]. When we use RAW-sockets we are sending unconnected packets .

• IP-headers of RAW sockets can be manually created by the programmer if the option IP\_HDRINCL is enabled [1]. This way, Raw sockets allow a programmer to implement new IP based protocols. If IP\_HDRINCL is not enabled the IP header will be generated automatically.

• When a RAW socket is created any IP based protocol can be specified. This results in a socket only receiving messages of the type of the specified protocol.

• If a programmer does not want to specify a protocol when creating a RAW-socket he can also use the IPPROTO\_RAW protocol (which implies that the headers will be created manually) . This way he can send any IP based protocol. However this socket is not able to receive any IP packets, to receive all IP based packets a PACKET-socket has to be used.

1. **Unix (FreeBSD, Mac OS X)**

Like Linux under Unix operating systems there are also RAW-sockets provided with the Unix kernel. RAW-sockets are however more restricted than under Linux. Access to the Data Link Layer is possible in Unix via the Berkeley Packet Filters (BPF) provided by the operating system

* 1. **RAW-sockets**

RAW-sockets are included in the socket-API in Unix like they are in Linux. In Unix there are different header files available and needed than in Linux, that makes some considerations necessary to keep code portable between these two operating systems [1]. Also there are some differences to the functionality.

Similar to Linux there are some restrictions to take into consideration:

• It is not possible to read packets for anything that has a handler (like TCP or UDP), but it is possible to read packets for other protocols like ICMP .

• For BSD and its ports it could be that the Packets are modified by the operating system before sending. For example with the release 10 the IP length is modified to the actual size of the IP header regardless of what is set by the programmer.

• Due to essential nature of header information for networking functionality and security reasons using RAWsockets requires root access permission .

• The ports of the Network layer are not endpoints anymore, since RAW-sockets work on the layers below . Filtering based on ports has to be done manually.

• The bind() and connect() functions are no longer necessary . bind() and connect() can still be used to define source address and target address to be entered automatically by the kernel .

• The listen() and accept() functions are no use at all since the Client-Server-Semantic is no longer present . When we use RAW-sockets we are sending unconnected packets.

**Programming with the APIs**

**RAW-sockets:**

The first API discussed here is the RAW-socket. It is available on Linux and Unix Systems.

RAW-sockets allow access to the Network (OSI layer 4) and Internet

(OSI layer -3) layer of the network stack.

The use of RAW-sockets is limited to processes with an effective user ID of 0 or the CAP\_NET\_RAW capability since they require root-access permissions.

**In the following we will start by giving a generic overview over the functions available and necessary to create and work with RAW-sockets. After that we show the structure of different protocol level write and read operations.**

**socket() :** Both the read and write to a RAW-socket require the socket to be created first. For the creation of a socket the same function as for normal sockets is used.

It is available in the header and has the following form.

int socket(int family, int type, int protocol)

**setsockopt() :** The setsockopt() function like the name implies can be used to change the options that are selected for the socket. The function can manipulate the options for different protocol levels such as IP or TCP, but also for the sockets level API by setting the level to SOL\_socket.

The function which is defined in the header has the following form .

int setsockopt(int sockfd, int level, int optname, const void \* optval, socklen\_t optlen)

**getsockopt() :** For retrieving an specified option of the socket the getsockopt() function defined in the header can be used .

int getsockopt(int sockfd, int level, int optname, void \* optval, socklen\_t \* optlen)

**bind()** : After creating a socket like discussed in the previous section 3.2.1 on page 15, we can bind the created socket to a specific address. Traditionally this is also called assigning a name to a socket. For RAW-sockets and PACKETsockets this is optional, but we use it to define the source address of our packets with it and also define from which network-interface we want to read packets. Other socket types might require to be bound to a specific address before they can be used. For binding the socket to an IP-address we can use the bind() function which is defined in the header and has the following form.

int bind(int sockfd, const struct sockaddr \* addr, socklen\_t addrlen)

**getsockname()** : To get the currently defined name of a socket the getsocketname() function can be used. It is defined in the and has the following form .

int getsocketname(int sockfd, struct sockaddr \* addr, socklen\_t \* addrlen)

**connect()** : This function can be used to initiates a connection to a specific destination host and is required for the write(), send(), read() and recv() functions. For connection based protocols like SOCK\_STREAM this function also will try to connect to the host on the other side [8]. For Datagram based protocols only the default destination is defined with this function [8]. To use this function the header (for the function) has to be included.

The function is called like this.

int connect(int sockfd, const struct sockaddr \* addr, socklen\_t addrlen)

**read()** : The read function works identical to the read() on a file. As already said the the socket can be connected to a specific host first by using the connect function as described in section 3.2.6 on the preceding page. For the read() function the definitions can be found in the header. The function has the following form [8].

int read(int fd, char \* Buff, int NumBytes)

**recv()** : The recv( function is another possibility to to retrieve data from a socket and does not require an address to be defined as well. As already said the the socket can be connected to a specific host first by using the connect() function as described in section 3.2.6 on the preceding page.To use the recv() function the headers (for the data types) and (for the function) have to be included.

The function is called like this.

ssize\_t recv(int sockfd, void \* buf, size\_t len, int flags)

**recvfrom()** : The recvfrom() function also allows us to define the address of a host we want to receive data from. To use this function the headers (for the data types) and (for the function) have to be included.

The function is called like this.

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

**write()** : Identical to how we use a the read() function to get data from a socket descriptor instead of a file descriptor, we can also use the write() function that is usually used for data output to a file for sending data over a socket. As already mentioned to use write() with a socket a valid destination address has to be provided to the socket first for it to work. The write() function is defined in and has the following from.

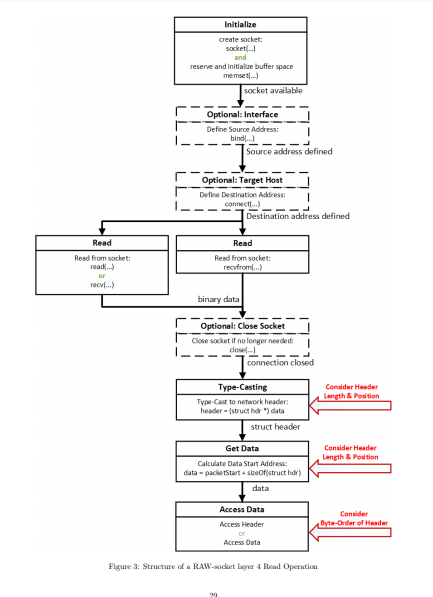
ssize\_t write(int fd, const void \* buf, size\_t count)

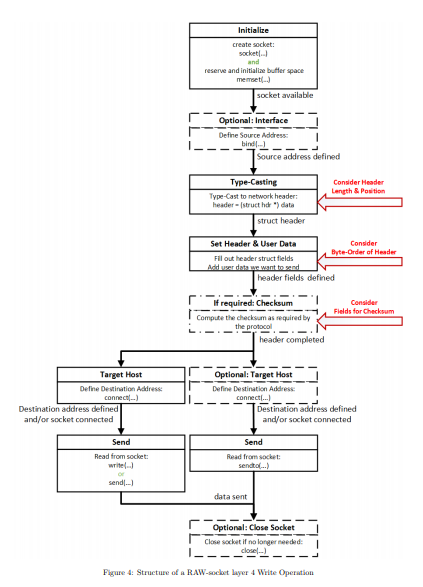
**send()** : The send() function is another function to send data over a socket. Like write() it does require that a destination address is specified first before it can be used, since it does require the socket to be in a connected state.There is no indication of a failure implicitly shown when using send(), only locally detected errors are indicated by returning -1 [8]. If send() is used to transmit messages, then it will block (if not set otherwise) if a message does not fit in the buffer. To use this function the headers (for the data types) and (for 22 the function) have to be included.

The function is called like this .

ssize\_t send(int sockfd, void \* buf, size\_t len, int flags)

**Layer 4**





**Layer 3:**

