A raw socket is a type of socket that allows access to the underlying transport provider. This topic focuses only on raw sockets and the IPv4 and IPv6 protocols. This is because most other protocols with the exception of ATM do not support raw sockets. To use raw sockets, an application needs to have detailed information on the underlying protocol being used.

Winsock service providers for the IP protocol may support a socket *type* of **SOCK\_RAW**. The Windows Sockets 2 provider for TCP/IP included on Windows supports this **SOCK\_RAW** socket type.

There are two basic types of such raw sockets:

* The first type uses a known protocol type written in the IP header that is recognized by a Winsock service provider. An example of the first type of socket is a socket for the ICMP protocol (IP protocol type = 1) or the ICMPv6 protocol (IP procotol type = 58).
* The second type allows any protocol type to be specified. An example of the second type would be an experimental protocol that is not directly supported by the Winsock service provider such as the Stream Control Transmission Protocol (SCTP).

**Determining if Raw Sockets are Supported**

If a Winsock service provider supports **SOCK\_RAW** sockets for the AF\_INET or AF\_INET6 address families, the socket type of **SOCK\_RAW** should be included in the [**WSAPROTOCOL\_INFO**](https://learn.microsoft.com/en-us/windows/win32/api/winsock2/ns-winsock2-wsaprotocol_infoa) structure returned by **[WSAEnumProtocols](https://learn.microsoft.com/en-us/windows/desktop/api/Winsock2/nf-winsock2-wsaenumprotocolsa)** function for one or more of the available transport providers.

The **iAddressFamily** member in the [**WSAPROTOCOL\_INFO**](https://learn.microsoft.com/en-us/windows/win32/api/winsock2/ns-winsock2-wsaprotocol_infoa) structure should specify AF\_INET or AF\_INET6 and the **iSocketType** member of the **WSAPROTOCOL\_INFO** structure should specify **SOCK\_RAW** for one of the transport providers.

The **iProtocol** member of the [**WSAPROTOCOL\_INFO**](https://learn.microsoft.com/en-us/windows/win32/api/winsock2/ns-winsock2-wsaprotocol_infoa) structure may be set to **IPROTO\_IP**. The **iProtocol** member of the **WSAPROTOCOL\_INFO** structure may also be set to zero if the service provider allows an application to use a **SOCK\_RAW** socket type for other network protocols other than the Internet Protocol for the address family.

The other members in the [**WSAPROTOCOL\_INFO**](https://learn.microsoft.com/en-us/windows/win32/api/winsock2/ns-winsock2-wsaprotocol_infoa) structure indicate other properties of the protocol support for **SOCK\_RAW** and indicate how a socket of **SOCK\_RAW** should be treated. These other members of the **WSAPROTOCOL\_INFO** for **SOCK\_RAW** normally specify that the protocol is connectionless, message-oriented, supports broadcast/multicast (the XP1\_CONNECTIONLESS, XP1\_MESSAGE\_ORIENTED, XP1\_SUPPORT\_BROADCAST, and XP1\_SUPPORT\_MULTIPOINT bits are set in the dwServiceFlags1 member), and can have a maximum message size of 65,467 bytes.

On Windows XP and later, the *NetSh.exe* command can be used to determine if raw sockets are supported. The following command run from a CMD window will display data from the Winsock catalog on the console:

**netsh winsock show catalog**

The output will include a list that contains some of the data from the [**WSAPROTOCOL\_INFO**](https://learn.microsoft.com/en-us/windows/win32/api/winsock2/ns-winsock2-wsaprotocol_infoa) structures supported on the local computer. Search for the term RAW/IP or RAW/IPv6 in the Description field to find those protocols that support raw sockets.

**Creating a Raw Socket**

To create a socket of type **SOCK\_RAW**, call the [**socket**](https://learn.microsoft.com/en-us/windows/desktop/api/Winsock2/nf-winsock2-socket) or **[WSASocket](https://learn.microsoft.com/en-us/windows/desktop/api/Winsock2/nf-winsock2-wsasocketa)** function with the *af* parameter (address family) set to AF\_INET or AF\_INET6, the *type* parameter set to **SOCK\_RAW**, and the *protocol* parameter set to the protocol number required. The *protocol* parameter becomes the protocol value in the IP header (SCTP is 132, for example).

**Note**

An application may not specify zero (0) as the *protocol* parameter for the [**socket**](https://learn.microsoft.com/en-us/windows/desktop/api/Winsock2/nf-winsock2-socket), **[WSASocket](https://learn.microsoft.com/en-us/windows/desktop/api/Winsock2/nf-winsock2-wsasocketa)**, and **[WSPSocket](https://learn.microsoft.com/en-us/windows/desktop/api/Ws2spi/nc-ws2spi-lpwspsocket)** functions if the *type* parameter is set to **SOCK\_RAW**.

Raw sockets offer the capability to manipulate the underlying transport, so they can be used for malicious purposes that pose a security threat. Therefore, only members of the Administrators group can create sockets of type SOCK\_RAW on Windows 2000 and later.

**Send and Receive Operations**

Once an application creates a socket of type **SOCK\_RAW**, this socket may be used to send and receive data. All packets sent or received on a socket of type **SOCK\_RAW** are treated as datagrams on an unconnected socket.

The following rules apply to the operations over **SOCK\_RAW** sockets:

* The **[sendto](https://learn.microsoft.com/en-us/windows/desktop/api/winsock/nf-winsock-sendto)** or **[WSASendTo](https://learn.microsoft.com/en-us/windows/desktop/api/Winsock2/nf-winsock2-wsasendto)** function is normally used to send data on a socket of type **SOCK\_RAW**. The destination address can be any valid address in the socket's address family, including a broadcast or multicast address. To send to a broadcast address, an application must have used **[setsockopt](https://learn.microsoft.com/en-us/windows/desktop/api/winsock/nf-winsock-setsockopt)** with SO\_BROADCAST enabled. Otherwise, **sendto** or **WSASendTo** will fail with the error code [WSAEACCES](https://learn.microsoft.com/en-us/windows/win32/winsock/windows-sockets-error-codes-2). For IP, an application can send to any multicast address (without becoming a group member).
* When sending IPv4 data, an application has a choice on whether to specify the IPv4 header at the front of the outgoing datagram for the packet. If the **IP\_HDRINCL** socket option is set to true for an IPv4 socket (address family of AF\_INET), the application must supply the IPv4 header in the outgoing data for send operations. If this socket option is false (the default setting), then the IPv4 header should not be in included the outgoing data for send operations.
* When sending IPv6 data, an application has a choice on whether to specify the IPv6 header at the front of the outgoing datagram for the packet. If the **IPV6\_HDRINCL** socket option is set to true for an IPv6 socket (address family of AF\_INET6), the application must supply the IPv6 header in the outgoing data for send operations. The default setting for this option is false. If this socket option is false (the default setting), then the IPv6 header should not be included in the outgoing data for send operations. For IPv6, there should be no need to include the IPv6 header. If information is available using socket functions, then the IPv6 header should not be included to avoid compatibility problems in the future. These issues are discussed in RFC 3542 published by the IETF. Using the **IPV6\_HDRINCL** socket option is not recommended and may be deprecated in future.
* The **[recvfrom](https://learn.microsoft.com/en-us/windows/desktop/api/winsock/nf-winsock-recvfrom)** or **[WSARecvFrom](https://learn.microsoft.com/en-us/windows/desktop/api/Winsock2/nf-winsock2-wsarecvfrom)** function is normally used to receive data on a socket of type **SOCK\_RAW**. Both of these functions have an option to return the source IP address where the packet was sent from. The received data is a datagram from an unconnected socket.
* For IPv4 (address family of AF\_INET), an application receives the IP header at the front of each received datagram regardless of the **IP\_HDRINCL** socket option.
* For IPv6 (address family of AF\_INET6), an application receives everything after the last IPv6 header in each received datagram regardless of the **IPV6\_HDRINCL** socket option. The application does not receive any IPv6 headers using a raw socket.
* Received datagrams are copied into all **SOCK\_RAW** sockets that satisfy the following conditions:
  + The protocol number specified in the *protocol* parameter when the socket was created should match the protocol number in the IP header of the received datagram.
  + If a local IP address is defined for the socket, it should correspond to the destination address as specified in the IP header of the received datagram. An application may specify the local IP address by calling the [**bind**](https://learn.microsoft.com/en-us/windows/desktop/api/winsock/nf-winsock-bind) function. If no local IP address is specified for the socket, the datagrams are copied into the socket regardless of the destination IP address in the IP header of the received datagram.
  + If a foreign address is defined for the socket, it should correspond to the source address as specified in the IP header of the received datagram. An application may specify the foreign IP address by calling the [**connect**](https://learn.microsoft.com/en-us/windows/desktop/api/Winsock2/nf-winsock2-connect) or **[WSAConnect](https://learn.microsoft.com/en-us/windows/desktop/api/Winsock2/nf-winsock2-wsaconnect)** function. If no foreign IP address is specified for the socket, the datagrams are copied into the socket regardless of the source IP address in the IP header of the received datagram.

It is important to understand that some sockets of type **SOCK\_RAW** may receive many unexpected datagrams. For example, a PING program may create a socket of type **SOCK\_RAW** to send ICMP echo requests and receive responses. While the application is expecting ICMP echo responses, all other ICMP messages (such as ICMP HOST\_UNREACHABLE) may also be delivered to this application. Moreover, if several **SOCK\_RAW** sockets are open on a computer at the same time, the same datagrams may be delivered to all the open sockets. An application must have a mechanism to recognize the datagrams of interest and to ignore all others. For a PING program, such a mechanism might include inspecting the received IP header for unique identifiers in the ICMP header (the application's process ID, for example).

**Note**

To use a socket of type **SOCK\_RAW** requires administrative privileges. Users running Winsock applications that use raw sockets must be a member of the Administrators group on the local computer, otherwise raw socket calls will fail with an error code of [**WSAEACCES**](https://learn.microsoft.com/en-us/windows/win32/winsock/windows-sockets-error-codes-2). On Windows Vista and later, access for raw sockets is enforced at socket creation. In earlier versions of Windows, access for raw sockets is enforced during other socket operations.

**Common Uses of Raw Sockets**

One common use of raw sockets are troubleshooting applications that need to examine IP packets and headers in detail. For example, a raw socket can be used with the SIO\_RCVALL IOCTL to enable a socket to receive all IPv4 or IPv6 packets passing through a network interface. For more information, see the [**SIO\_RCVALL**](https://learn.microsoft.com/en-us/windows/win32/winsock/sio-rcvall) reference.

**Limitations on Raw Sockets**

On Windows 7, Windows Vista, Windows XP with Service Pack 2 (SP2), and Windows XP with Service Pack 3 (SP3), the ability to send traffic over raw sockets has been restricted in several ways:

* TCP data cannot be sent over raw sockets.
* UDP datagrams with an invalid source address cannot be sent over raw sockets. The IP source address for any outgoing UDP datagram must exist on a network interface or the datagram is dropped. This change was made to limit the ability of malicious code to create distributed denial-of-service attacks and limits the ability to send spoofed packets (TCP/IP packets with a forged source IP address).
* A call to the [**bind**](https://learn.microsoft.com/en-us/windows/desktop/api/winsock/nf-winsock-bind) function with a raw socket for the IPPROTO\_TCP protocol is not allowed.

**Note**

The [**bind**](https://learn.microsoft.com/en-us/windows/desktop/api/winsock/nf-winsock-bind) function with a raw socket is allowed for other protocols (IPPROTO\_IP, IPPROTO\_UDP, or IPPROTO\_SCTP, for example).

These above restrictions do not apply to Windows Server 2008 R2, Windows Server 2008 , Windows Server 2003, or to versions of the operating system earlier than Windows XP with SP2.

**Note**

The Microsoft implementation of TCP/IP on Windows is capable of opening a raw UDP or TCP socket based on the above restrictions. Other Winsock providers may not support the use of raw sockets.

There are further limitations for applications that use a socket of type **SOCK\_RAW**. For example, all applications listening for a specific protocol will receive all packets received for this protocol. This may not be what is desired for multiple applications using a protocol. This is also not suitable for high-performance applications. To get around these issues, it may be required to write a Windows network protocol driver (device driver) for the specific network protocol. On Windows Vista and later, Winsock Kernel (WSK), a new transport-independent kernel mode Network Programming Interface can be used to write a network protocol driver. On Windows Server 2003 and earlier, a Transport Driver Interface (TDI) provider and a Winsock helper DLL can be written to support the network protocol. The network protocol would then be added to the Winsock catalog as a supported protocol. This allows multiple applications to open sockets for this specific protocol and the device driver can keep track of which socket receives specific packets and errors. For information on writing a network protocol provider, see the sections on WSK and TDI in the Windows Driver Kit (WDK).

Applications also need to be aware of the impact that firewall settings may have on sending and receiving packets using raw sockets.

A Guide to Using Raw Sockets

By

[**Subodh Saxena**](https://www.opensourceforu.com/author/subodh-saxena/)

 -

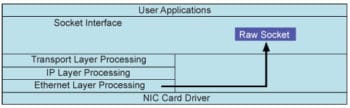
March 21, 2015

[22](https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/#comments)

197874

*In this tutorial, let us take a look at how raw sockets can be used to receive data packets and send those packets to specific user applications, bypassing the normal TCP/IP protocols.*

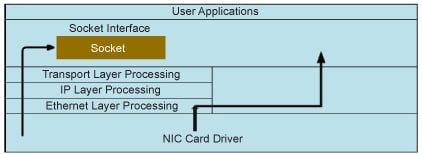
If you have no knowledge of the Linux kernel, yet are interested in the contents of network packets, raw sockets are the answer. Raw sockets are used to receive raw packets. This means packets received at the Ethernet layer will directly pass to the raw socket. Stating it precisely, a raw socket bypasses the normal TCP/IP processing and sends the packets to the specific user application (see Figure 1).

Figure 1: Graphical demonstration of a raw socket

A raw socket vs other sockets

Other sockets like stream sockets and data gram sockets receive data from the transport layer that contains no headers but only the payload. This means that there is no information about the source IP address and MAC address. If applications running on the same machine or on different machines are communicating, then they are only exchanging data.

The purpose of a raw socket is absolutely different. A raw socket allows an application to directly access lower level protocols, which means a raw socket receives un-extracted packets (see Figure 2). There is no need to provide the port and IP address to a raw socket, unlike in the case of stream and datagram sockets.

[](https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/figure-2-104/)Figure 2: Graphical demonstration of how a raw socket works compared to other sockets

Network packets and packet sniffers

When an application sends data into the network, it is processed by various network layers. Before sending data, it is wrapped in various headers of the network layer. The wrapped form of data, which contains all the information like the source and destination address, is called a network packet (see Figure 3). According to Ethernet protocols, there are various types of network packets like Internet Protocol packets, Xerox PUP packets, Ethernet Loopback packets, etc. In Linux, we can see all protocols in the *if\_ether.h* header file (see Figure 4).

[Figure 3 Generic Network Packet Representation](https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/figure-3-generic-network-packet-representation/)Figure 3: A generic representation of a network packet[Figure 4 Network Packet for IP Protocol](https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/figure-4-network-packet-for-ip-protocol/)Figure 4: Network Packet for internet Protocol

When we connect to the Internet, we receive network packets, and our machine extracts all network layer headers and sends data to a particular application. For example, when we type *www.google.com* in our browser, we receive packets sent from Google, and our machine extracts all the headers of the network layer and gives the data to our browser.

By default, a machine receives those packets that have the same destination address as that of the machine, and this mode is called the non-promiscuous mode. But if we want to receive all the packets, we have to switch into the promiscuous mode. We can go into the promiscuous mode with the help of*ioctls.*

If we are interested in the contents or the structure of the headers of different network layers, we can access these with the help of a packet sniffer. There are various packet sniffers available for Linux, like Wireshark. There is a command line sniffer called *tcpdump,* which is also a very good packet sniffer. And if we want to make our own packet sniffer, it can easily be done if we know the basics of C and networking.

A packet sniffer with a raw socket

To develop a packet sniffer, you first have to open a raw socket. Only processes with an effective user ID of 0 or the*CAP\_NET\_RAW* capability are allowed to open raw sockets. So, during the execution of the program, you have to be the root user.

Opening a raw socket

To open a socket, you have to know three things  the socket family, socket type and protocol. For a raw socket, the socket family is *AF\_PACKET*, the socket type is *SOCK\_RAW* and for the protocol, see the if\_ether.h header file. To receive all packets, the macro is *ETH\_P\_ALL* and to receive IP packets, the macro is *ETH\_P\_IP* for the protocol field.

|  |
| --- |
| int sock\_r;  sock\_r=socket(AF\_PACKET,SOCK\_RAW,htons(ETH\_P\_ALL));  **if**(sock\_r<0)  {  printf(error **in** socket\n);  **return** -1;  } |

Reception of the network packet

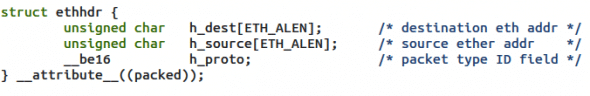
After successfully opening a raw socket, its time to receive network packets, for which you need to use the *recvfrom api.* We can also use the *recv api*. But recvfrom provides additional information.

|  |
| --- |
| unsigned char \*buffer = (unsigned char \*) malloc(65536); //to receive data  memset(buffer,0,65536);  struct sockaddr saddr;  int saddr\_len = sizeof (saddr);    //Receive a network packet and copy **in** to buffer  buflen=recvfrom(sock\_r,buffer,65536,0,&saddr,(socklen\_t \*)&saddr\_len);  **if**(buflen<0)  {  printf(error **in** reading recvfrom **function**\n);  **return** -1;  } |

In *saddr*, the underlying protocol provides the source address of the packet.

Extracting the Ethernet header

Now that we have the network packets in our buffer, we will get information about the Ethernet header. The Ethernet header contains the physical address of the source and destination, or the MAC address and protocol of the receiving packet. The *if\_ether.h* header contains the structure of the Ethernet header (see Figure 5).

[](https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/figure-5-structure-of-ethernet-header/)Figure 5: Structure of Ethernet header

Now, we can easily access these fields:

|  |
| --- |
| struct ethhdr \*eth = (struct ethhdr \*)(buffer);  printf(\nEthernet Header\n);  printf(\t|-Source Address : %.2X-%.2X-%.2X-%.2X-%.2X-%.2X\n,eth->h\_source[0],eth->h\_source[1],eth->h\_source[2],eth->h\_source[3],eth->h\_source[4],eth->h\_source[5]);  printf(\t|-Destination Address : %.2X-%.2X-%.2X-%.2X-%.2X-%.2X\n,eth->h\_dest[0],eth->h\_dest[1],eth->h\_dest[2],eth->h\_dest[3],eth->h\_dest[4],eth->h\_dest[5]);  printf(\t|-Protocol : %d\n,eth->h\_proto); |

*h\_proto* gives information about the next layer. If you get 0x800 *(ETH\_P\_IP)*, it means that the next header is the IP header. Later, we will consider the next header as the IP header.

**Note 1:** *The physical address is 6 bytes*.

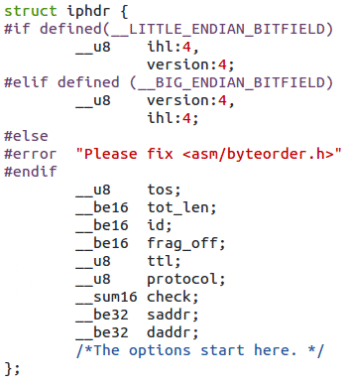
**Note 2:** *We can also direct the output to a file for better understanding.*

|  |
| --- |
| fprintf(log\_txt,\t|-Source Address : %.2X-%.2X-%.2X-%.2X-%.2X-%.2X\n,eth->h\_source[0],eth->h\_source[1],eth->h\_source[2],eth->h\_source[3],eth->h\_source[4],eth->h\_source[5]); |

Use*fflush* to avoid the input-output buffer problem when writing into a file.

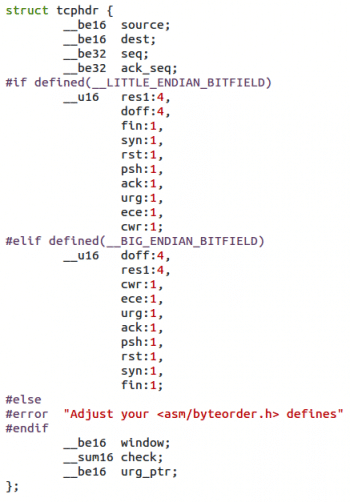
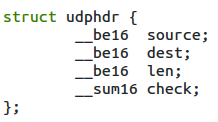
Extracting the IP header

The IP layer gives various pieces of information like the source and destination IP address, the transport layer protocol, etc. The structure of the IP header is defined in the*ip.h header file* (see Figure 6).

[](https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/figure-6-structure-of-ip-header/)Figure 6: Structure of IP Header

Now, to get this information, you need to increment your buffer pointer by the size of the Ethernet header because the IP header comes after the Ethernet header:

|  |
| --- |
| unsigned short iphdrlen;  struct iphdr \*ip = (struct iphdr\*)(buffer + sizeof(struct ethhdr));  memset(&source, 0, sizeof(source));  source.sin\_addr.s\_addr = ip->saddr;  memset(&dest, 0, sizeof(dest));  dest.sin\_addr.s\_addr = ip->daddr;    fprintf(log\_txt, \t|-Version : %d\n,(unsigned int)ip->version);    fprintf(log\_txt , \t|-Internet Header Length : %d DWORDS or %d Bytes\n,(unsigned int)ip->ihl,((unsigned int)(ip->ihl))\*4);    fprintf(log\_txt , \t|-Type Of Service : %d\n,(unsigned int)ip->tos);    fprintf(log\_txt , \t|-Total Length : %d Bytes\n,ntohs(ip->tot\_len));    fprintf(log\_txt , \t|-Identification : %d\n,ntohs(ip->id));    fprintf(log\_txt , \t|-Time To Live : %d\n,(unsigned int)ip->ttl);    fprintf(log\_txt , \t|-Protocol : %d\n,(unsigned int)ip->protocol);    fprintf(log\_txt , \t|-Header Checksum : %d\n,ntohs(ip->check));    fprintf(log\_txt , \t|-Source IP : %s\n, inet\_ntoa(source.sin\_addr));    fprintf(log\_txt , \t|-Destination IP : %s\n,inet\_ntoa(dest.sin\_addr)); |

[](https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/figure-7-structure-of-tcp-header/)Figure 7: Structure of TCP Header[](https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/figure-8-structure-of-udp-header/)Figure 8: Structure of UDP Header

The transport layer header

There are various transport layer protocols. Since the underlying header was the IP header, we have various IP or Internet protocols. You can see these protocols in the */etc/protocls file.* The TCP and UDP protocol structures are defined in tcp.h and udp.h respectively. These structures provide the port number of the source and destination. With the help of the port number, the system gives data to a particular application (see Figures 7 and 8).  
The size of the IP header varies from 20 bytes to 60 bytes. We can calculate this from the IP header field or IHL. IHL means Internet Header Length (IHL), which is the number of 32-bit words in the header. So we have to multiply the IHL by 4 to get the size of the header in bytes:

|  |
| --- |
| struct iphdr \*ip = (struct iphdr \*)( buffer + sizeof(struct ethhdr) );  /\* getting actual size of IP header\*/  iphdrlen = ip->ihl\*4;  /\* getting pointer to udp header\*/  struct tcphdr \*udp=(struct udphdr\*)(buffer + iphdrlen + sizeof(struct ethhdr)); |

We now have the pointer to the UDP header. So lets check some of its fields.

**Note:***If your machine is little endian, you have to use ntohs because the network uses the big endian scheme.*

|  |
| --- |
| fprintf(log\_txt , \t|-Source Port : %d\n , ntohs(udp->source));  fprintf(log\_txt , \t|-Destination Port : %d\n , ntohs(udp->dest));  fprintf(log\_txt , \t|-UDP Length : %d\n , ntohs(udp->len));  fprintf(log\_txt , \t|-UDP Checksum : %d\n , ntohs(udp->check)); |

Similarly, we can access the TCP header field.

Extracting data

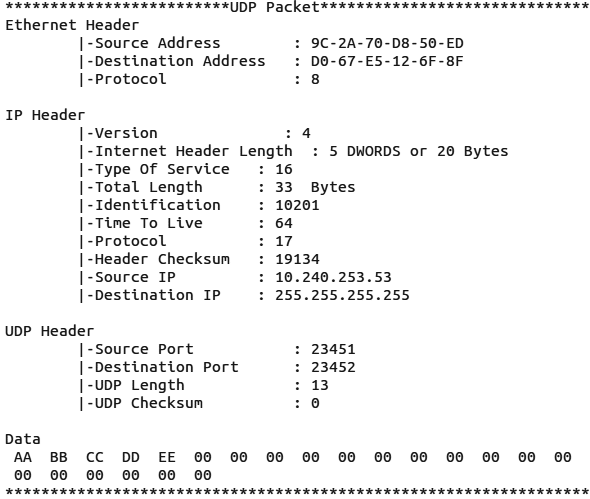
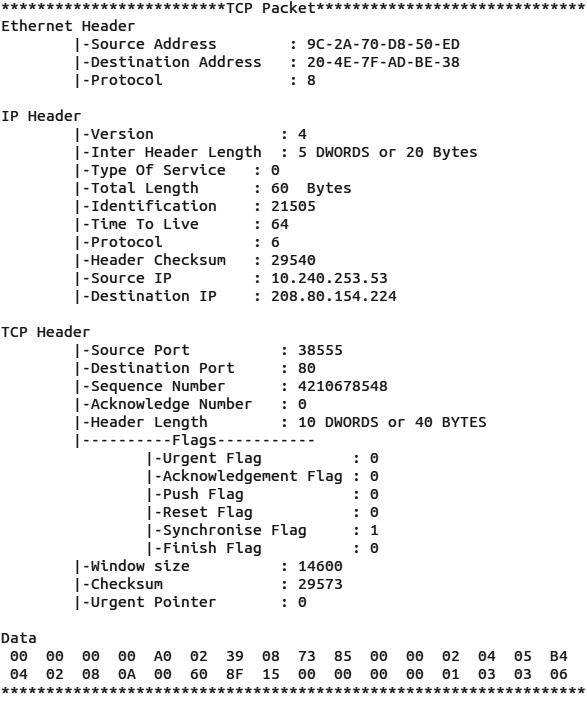
After the transport layer header, there is data payload remaining. For this, we will move the pointer to the data, and then print.

|  |
| --- |
| unsigned char \* data = (buffer + iphdrlen + sizeof(struct ethhdr) + sizeof(struct udphdr)); |

Now, lets print data, and for better representation, let us print 16 bytes in a line.

|  |
| --- |
| int remaining\_data = buflen - (iphdrlen + sizeof(struct ethhdr) + sizeof(struct udphdr));    **for**(i=0;i<remaining\_data;i++)  {  **if**(i!=0 && i%16==0)  fprintf(log\_txt,\n);  fprintf(log\_txt, %.2X ,data[i]);  } |

When you receive a packet, it will look like whats shown is Figures 9 and 10.

[](https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/figure-9-udp-packet/)Figure 9: UDP Packet[](https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/figure-10-tcp-packet/)Figure 10: TCP Packet

Sending packets with a raw socket

To send a packet, we first have to know the source and destination IP addresses as well as the MAC address. Use your friends *MAC & IP* address as the destination IP and MAC address. There are two ways to find out your IP address and MAC address:

1. Enter*ifconfig* and get the IP and MAC for a particular interface.
2. Enter*ioctl* and get the IP and MAC.

The second way is more efficient and will make your program machine-independent, which means you should not enter*ifconfig* in each machine.

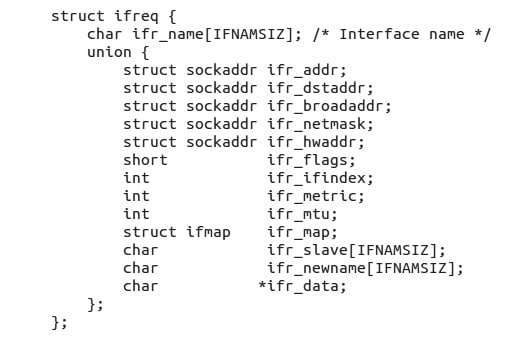
Opening a raw socket

To open a raw socket, you have to know three fields of socket *API — Family- AF\_PACKET, Type- SOCK\_RAW* and for the protocol, lets use *IPPROTO\_RAW* because we are trying to send an IP packet.*IPPROTO\_RAW* macro is defined in the in.h header file:

|  |
| --- |
| sock\_raw=socket(AF\_PACKET,SOCK\_RAW,IPPROTO\_RAW);  **if**(sock\_raw == -1)  printf(error **in** socket); |

What is struct ifreq?

Linux supports some standard ioctls to configure network devices. They can be used on any sockets file descriptor, regardless of the family or type. They pass an ifreq structure, which means that if you want to know some information about the network, like the interface index or interface name, you can use ioctl and it will fill the value of the*ifreq* structure passed as a third argument. In short, the *ifreq* structure is a way to get and set the network configuration. It is defined in the if.h header file or you can check the man page of *netdevice* (see Figure 11).

[](https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/figure-11-structure-of-ifreq/)Figure 11: Structure of *ifreq*Figure 12: Graphical representation of packets with their structure and payload

Getting the index of the interface to send a packet

There may be various interfaces in your machine like*loopback, wired interface* and *wireless interface*. So you have to decide the interface through which we can send our packet. After deciding on the interface, you have to get the index of that interface. For this, first give the name of the interface by setting the field i*fr\_name* of *ifreq* structure, and then use ioctl. Then use the *SIOCGIFINDEX* macro defined in sockios.h and you will receive the index number in the ifreq structure:

|  |
| --- |
| struct ifreq ifreq\_i;  memset(&ifreq\_i,0,sizeof(ifreq\_i));  strncpy(ifreq\_i.ifr\_name,wlan0,IFNAMSIZ-1); //giving name of Interface    **if**((ioctl(sock\_raw,SIOCGIFINDEX,&ifreq\_i))<0)  printf(error **in** index ioctl reading);//getting Index Name    printf(index=%d\n,ifreq\_i.ifr\_ifindex); |

Getting the MAC address of the interface

Similarly, you can get the MAC address of the interface, for which you need to use the SIOCGIFHWADDR macro to ioctl:

|  |
| --- |
| struct ifreq ifreq\_c;  memset(&ifreq\_c,0,sizeof(ifreq\_c));  strncpy(ifreq\_c.ifr\_name,wlan0,IFNAMSIZ-1);//giving name of Interface    **if**((ioctl(sock\_raw,SIOCGIFHWADDR,&ifreq\_c))<0) //getting MAC Address  printf(error **in** SIOCGIFHWADDR ioctl reading); |

Getting the IP address of the interface

For this, use the SIOCGIFADDR macro:

|  |
| --- |
| struct ifreq ifreq\_ip;  memset(&ifreq\_ip,0,sizeof(ifreq\_ip));  strncpy(ifreq\_ip.ifr\_name,wlan0,IFNAMSIZ-1);//giving name of Interface  **if**(ioctl(sock\_raw,SIOCGIFADDR,&ifreq\_ip)<0) //getting IP Address  {  printf(error **in** SIOCGIFADDR \n);  } |

Constructing the Ethernet header

After getting the index, as well as the MAC and IP addresses of an interface, its time to construct the Ethernet header. First, take a buffer in which you will place all information like the Ethernet header, IP header, UDP header and data. That buffer will be your packet.

|  |
| --- |
| sendbuff=(unsigned char\*)malloc(64); // increase **in** **case** of more data  memset(sendbuff,0,64); |

To construct the Ethernet header, fill all the fields of the ethhdr structure:

|  |
| --- |
| struct ethhdr \*eth = (struct ethhdr \*)(sendbuff);    eth->h\_source[0] = (unsigned char)(ifreq\_c.ifr\_hwaddr.sa\_data[0]);  eth->h\_source[1] = (unsigned char)(ifreq\_c.ifr\_hwaddr.sa\_data[1]);  eth->h\_source[2] = (unsigned char)(ifreq\_c.ifr\_hwaddr.sa\_data[2]);  eth->h\_source[3] = (unsigned char)(ifreq\_c.ifr\_hwaddr.sa\_data[3]);  eth->h\_source[4] = (unsigned char)(ifreq\_c.ifr\_hwaddr.sa\_data[4]);  eth->h\_source[5] = (unsigned char)(ifreq\_c.ifr\_hwaddr.sa\_data[5]);    /\* filling destination mac. DESTMAC0 to DESTMAC5 are macro having octets of mac address. \*/  eth->h\_dest[0] = DESTMAC0;  eth->h\_dest[1] = DESTMAC1;  eth->h\_dest[2] = DESTMAC2;  eth->h\_dest[3] = DESTMAC3;  eth->h\_dest[4] = DESTMAC4;  eth->h\_dest[5] = DESTMAC5;    eth->h\_proto = htons(ETH\_P\_IP); //means next header will be IP header    /\* end of ethernet header \*/  total\_len+=sizeof(struct ethhdr); |

Constructing the IP header

To construct the IP header, increment sendbuff by the size of the Ethernet header and fill each field of the*iphdr* structure. Data after the IP header is called the payload for the IP header and, in the same way, data after the Ethernet header is called the payload for the Ethernet header. In the IP header, there is a field called Total Length, which contains the size of the IP header plus the payload. To know the size of the payload of the IP header, you must know the size of the UDP header and the UDP payload. So, some field of the *iphdr structure* will get the value after filling the UDP header field.

|  |
| --- |
| struct iphdr \*iph = (struct iphdr\*)(sendbuff + sizeof(struct ethhdr));  iph->ihl = 5;  iph->version = 4;  iph->tos = 16;  iph->id = htons(10201);  iph->ttl = 64;  iph->protocol = 17;  iph->saddr = inet\_addr(inet\_ntoa((((struct sockaddr\_in \*)&(ifreq\_ip.ifr\_addr))->sin\_addr)));  iph->daddr = inet\_addr(destination\_ip); // put destination IP address    total\_len += sizeof(struct iphdr); |

Construct the UDP header

Constructing the UDP header is very similar to constructing the IP header. Assign values to the fields of the *udphdr* structure. For this, increment the *sendbuff* pointer by the size of the Ethernet and the IP headers.

|  |
| --- |
| struct udphdr \*uh = (struct udphdr \*)(sendbuff + sizeof(struct iphdr) + sizeof(struct ethhdr));    uh->source = htons(23451);  uh->dest = htons(23452);  uh->check = 0;    total\_len+= sizeof(struct udphdr); |

Like the IP header, the UDP also has the field len, which contains the size of the UDP header and its payload. So, first, you have to know the UDP payload, which is the actual data that will be sent.

Adding data or the UDP payload

We can send any data:

|  |
| --- |
| sendbuff[total\_len++] = 0xAA;  sendbuff[total\_len++] = 0xBB;  sendbuff[total\_len++] = 0xCC;  sendbuff[total\_len++] = 0xDD;  sendbuff[total\_len++] = 0xEE; |

Filling the remaining fields of the IP and UDP headers

We now have the total\_len pointer and with the help of this, we can fill the remaining fields of the IP and UDP headers:

|  |
| --- |
| uh->len = htons((total\_len - sizeof(struct iphdr) - sizeof(struct ethhdr)));  //UDP length field  iph->tot\_len = htons(total\_len - sizeof(struct ethhdr));  //IP length field |

The IP header checksum

There is one more field remaining in the IP header check, which is used to have a checksum. A checksum is used for error checking of the header.

When the packet arrives at the router, it calculates the checksum, and if the calculated checksum does not match with the checksum field of the header, the router will drop the packet; and if it matches, the router will decrement the time to the live field by one, and forward it.

To calculate the checksum, sum up all the 16-bit words of the IP header and if there is any carry, add it again to get a 16-bit word. After this, find the complement of 1s and that is our checksum. To check whether our checksum is correct, use the above algorithm.

|  |
| --- |
| unsigned short checksum(unsigned short\* buff, int \_16bitword)  {  unsigned long sum;  **for**(sum=0;\_16bitword>0;\_16bitword--)  sum+=htons(\*(buff)++);  sum = ((sum >> 16) + (sum & 0xFFFF));  sum += (sum>>16);  **return** (unsigned short)(~sum);  }    iph->check = checksum((unsigned short\*)(sendbuff + sizeof(struct ethhdr)), (sizeof(struct iphdr)/2)); |

Sending the packet

Now we have our packet but before sending it, lets fill the *sockaddr\_ll* structure with the destination MAC address:

|  |
| --- |
| struct sockaddr\_ll sadr\_ll;  sadr\_ll.sll\_ifindex = ifreq\_i.ifr\_ifindex; // index of interface  sadr\_ll.sll\_halen = ETH\_ALEN; // length of destination mac address  sadr\_ll.sll\_addr[0] = DESTMAC0;  sadr\_ll.sll\_addr[1] = DESTMAC1;  sadr\_ll.sll\_addr[2] = DESTMAC2;  sadr\_ll.sll\_addr[3] = DESTMAC3;  sadr\_ll.sll\_addr[4] = DESTMAC4;  sadr\_ll.sll\_addr[5] = DESTMAC5; |

And now its time to send it, for which lets use the *sendto api:*

|  |
| --- |
| send\_len = sendto(sock\_raw,sendbuff,64,0,(const struct sockaddr\*)&sadr\_ll,sizeof(struct sockaddr\_ll));  **if**(send\_len<0)  {  printf(error **in** sending....sendlen=%d....errno=%d\n,send\_len,errno);  **return** -1;    } |

How to run the program

Go to*root user*, then compile and run your program in a machine. And in another machine, or in your destination machine, run the packet sniffer program as the root user and analyse the data that you are sending.

What to do next

We made a packet sniffer as well as a packet sender, but this is a user space task. Now lets try the same things in kernel space. For this, try to understand *struct sk\_buff* and make a module that can perform the same things in kernel space.

Raw Sockets in IPv6

21 Oct 2017 in [**IPv6**](https://labs.apnic.net/index.php/category/ipv6/) by Geoff Huston

Among many other functions performed by a computer’s operating system, there is typically an interface to a shared local network protocol engine. This means that applications that run within the operating system’s environment don’t need to implement their own network protocol engine, as they can make use of a shared common interface to the underlying network protocol engine via a simple standard interface. In Unix, the commonly used API to the underlying communications system is via the socket routines. Sockets creates an abstract model of the underlying network by mimicking a simple peripheral device, and once the application connects to a network socket (akin to opening a virtual connection to a remote host), it can then use the network through conventional read and write commands on that socket. Using the appropriate socket abstraction, the details about managing the TCP or UDP packet headers and the common IP layer is managed by the operating system’s protocol drivers, and is largely hidden from the application.

As part of a measurement experiment, we wanted an implementation of an IPv6 UDP server and a TCP server that generated fragmented IPv6 packets. However, as an added condition, we wanted the application to directly control the packet fragmentation function. The conventional standard socket interface masks any visibility to the underlying packet transactions, and therefore cannot be used for this experiment. There is a specialised socket option that allows an application to interact directly with the underlying communications driver and read and write IP datagrams without having the packets processed by the operating system’s IP protocol drivers. This is the Raw Socket option in IPv4, and an example of opening such a socket is shown in this C code snippet:

/\*

\* open\_raw\_socket

\*

\* open a raw socket interface into the kernel

\*/

void

open\_raw\_socket()

{

const int on = 1 ;

/\* create the raw socket via the socket call\*/

if ((sock\_fd = socket(AF\_INET, SOCK\_RAW, IPPROTO\_TCP)) < 0) {

perror("socket() error");

exit(EXIT\_FAILURE);

}

/\* inform the kernel the IP header is already attached via a socket option \*/

if (setsockopt(sock\_fd, IPPROTO\_IP, IP\_HDRINCL, &on, sizeof(on)) < 0) {

perror("setsockopt() error");

exit(EXIT\_FAILURE);

}

}

This code snippet is written to use IPv4. A shift to IPv6 does not entail a lot of code changes. It's a case of changing the protocol specifier from AF\_INET to AF\_INET6, expanding the size of the IP address data structures to 128 bits and there is not that much else to do. But this is not the case for raw sockets. Here IPv6 is indeed very different, as noted in RFC 3542:

[The] difference from IPv4 raw sockets is that complete packets  
(that is, IPv6 packets with extension headers) cannot be sent or  
received using the IPv6 raw sockets API.  
…  
When writing to a raw socket the kernel will automatically fragment  
the packet if its size exceeds the path MTU, inserting the required  
fragment headers.  
…  
Most IPv4 implementations give special treatment to a raw socket  
created with a third argument to socket() of IPPROTO\_RAW, whose value  
is normally 255, to have it mean that the application will send down  
complete packets including the IPv4 header. (Note: This feature was  
added to IPv4 in 1988 by Van Jacobson to support traceroute, allowing  
a complete IP header to be passed by the application, before the  
IP\_HDRINCL socket option was added.) We note that IPPROTO\_RAW has no  
special meaning to an IPv6 raw socket (and the IANA currently  
reserves the value of 255 when used as a next-header field).

*RFC 3542, “Advanced Sockets Application Program Interface (API) for IPv6”, W. Stevens et al, May 2003.*

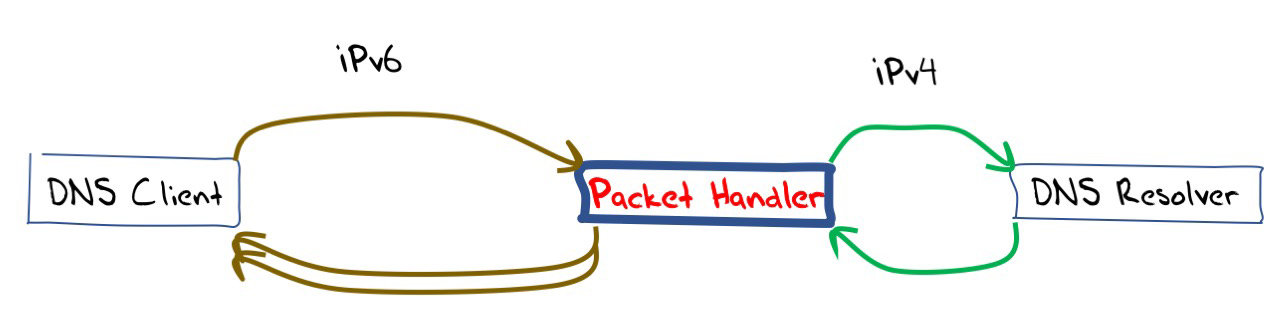
What we want to do in this measurement experiment, namely use an IPv6 raw socket that allows the application to have direct control of the entirety of the IPv6 packet header, including fragmentation handling, is not directly supported using the IPv6 raw IP socket interface.

But there is another approach that can be used. Sockets can extend one further level down in the protocol stack, and connect directly to the network interface rather than to a network protocol engine. This form of raw socket requires the application to write complete packets to the socket interface, including the media layer framing (such as the Ethernet headers), the IP level packet header, the transport protocol header, as well as any payload. If we want to perform explicit control over the generation of IPv6 Extension Headers to support IPv6 packet fragmentation control at the application level, then this media level socket interface looks like a viable approach.

In this article, I will describe how we used this raw socket interface in IPv6 to generate a UDP-based DNS server and a TCP-based HTTP(S) server that allowed the application to exercise direct control over packet fragmentation.

A Raw Socket UDP DNS Server

The aim of this packet handler was to provide a font end to a conventional DNS server. Incoming IPv6 UDP DNS queries are passed to a conventional “back end” DNS resolver. UDP responses from the “back end” DNS resolver are fragmented into at least two IPv6 packets, ensuring that no packet is larger than 512 octets, and passed back to the original sender.

*[](http://www.potaroo.net/ispcol/2017-10/raw-fig1.png)  
Figure 1 – UDP Packet Handler*

Let’s now look at the details of the code for this packet handler. The first part of the packet processor is a conventional UDP server listening port that listens on IPv6 to incoming packets addressed to the local port 53.

/\* Create a IPv6 datagram socket and associate it with the variable sockfd\*/

sockfd = socket(AF\_INET6,SOCK\_DGRAM,17);

if (sockfd < 0) {

perror("Socket");

exit(EXIT\_FAILURE);

}

/\* The local IPv6 address used to listen is stored in the variable host

This is converted to an internal representation of the IPv6 address \*/

if (((status = inet\_pton(AF\_INET6,host,&listen.sin6\_addr))) <= 0) {

if (!status)

fprintf(stderr, "Not in presentation format");

else

perror("inet\_pton");

exit(EXIT\_FAILURE);

}

/\* set this address into the socket structure and use port 53 (DNS) \*/

listen.sin6\_family = AF\_INET6 ;

listen.sin6\_port = htons(53);

/\* now bind the socket to the local IPv6 address and port 53 \*/

if (bind(sockfd,(const struct sockaddr \*) &listen,(sizeof listen))) {

perror("bind") ;

exit(EXIT\_FAILURE) ;

}

/\* we can now set up the listener – the source address and source port of

the incoming UDP packet is stored in the cliaddr struct \*/

addrlen= sizeof \*cliaddr ;

cliaddr=malloc(addrlen);

len=addrlen;

for ( ;; ) { /\* do forever \*/

if ((rc = recvfrom(sockfd, buf, MAXBUF,0,(struct sockaddr \*)cliaddr, &len)) < 0 ) {

printf("server error: errno %d\n",errno);

perror("reading datagram");

exit(1);

}

/\* at this stage the client’s address and port is stored in cliaddr, and the DNS query

is stored in buf, with length rc – we can pass this DNS query to the back end \*/

The next part of the code is also quite conventional. It takes the original DNS query, which is the payload returned by the recvfrom() call, and passes it to a DNS resolver without alteration. This example code is synchronous for simplicity, so the routine to query the back end DNS resolver will wait for the DNS resolver’s response before returning. The code also uses IPv4 for the connection to the DNS server.

/\* take an incoming UDP DNS query and flick it to a back-end DNS processor

and collect the UDP response \*/

/\* open a UDP socket to the DNS server at address serveraddr4 \*/

if ((dns\_sockfd = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0) {

perror("UDP slave socket() error") ;

return 1;

}

if (connect(dns\_sockfd, (struct sockaddr \*) &serveraddr4, sizeof(serveraddr4)) < 0) {

perror("UDP V4 connect error") ;

return 1;

}

/\* send the DNS query to the server \*/

i = write(sockfd, query, data\_size) ;

if (i < 0) {

perror("ERROR writing to socket");

return 1;

}

/\* collect the server's reply \*/

dns\_response.len = recvfrom(dns\_sockfd, dns\_response.buf, MAXBUF, 0,

(struct sockaddr \*) &serveraddr4, &serverlen);

if (dns\_response.len < 0) {

perror("UDP recvfrom");

return 1;

}

close(dns\_sockfd) ;

This is the simplest approach, but obviously not the most efficient. This code could be made asynchronous by using a non-blocking read calls on the socket, but for the sake of simplicity we will use the synchronous call to the socket.

The last part of the packet handler is the processing of IPv6 outbound UDP packets directed back to the original client, we here we will use a raw Ethernet socket interface.

The first part of this code opens a raw socket to in the interface named by the variable interface. In order to construct Ethernet packets, we need to get the Ethernet MAC address of the interface, which is performed by an ioctl call on a raw socket interface. We also use the if\_nametoindex() call to get the local index value of the named interface. We can then open a raw Ethernet socket on the named interface.

/\* Get a socket descriptor to look up interface \*/

if ((sd = socket (PF\_PACKET, SOCK\_RAW, htons (ETH\_P\_ALL))) < 0) {

perror("socket() failed to get socket descriptor for using ioctl() ");

exit(EXIT\_FAILURE);

}

/\* Use an ioctl() to look up interface name and get its MAC address \*/

/\* clear the ifr variable \*/

memset(&ifr, 0, sizeof (ifr));

/\* write in the name of the interface: held in variable interface \*/

snprintf(ifr.ifr\_name, sizeof (ifr.ifr\_name), "%s", interface);

if (ioctl(sd, SIOCGIFHWADDR, &ifr) < 0) {

perror("ioctl() failed to get source MAC address ");

exit (EXIT\_FAILURE);

}

/\* done! \*/

close(sd);

/\* Copy source MAC address into src\_mac \*/

memcpy(src\_mac, ifr.ifr\_hwaddr.sa\_data, 6 \* sizeof (uint8\_t));

/\* Find interface index from interface name using the call if\_nametoindex()

and store the index value in struct sockaddr\_ll device (which will be

used as an argument of sendto() of the subsequent output call \*/

memset(&device, 0, sizeof (device));

if ((device.sll\_ifindex = if\_nametoindex(interface)) == 0) {

perror ("if\_nametoindex() failed to obtain interface index ");

exit (EXIT\_FAILURE);

}

/\* now copy over the mac address \*/

device.sll\_family = AF\_PACKET;

memcpy(device.sll\_addr, src\_mac, 6 \* sizeof (uint8\_t));

device.sll\_halen = 6;

/\* now we are ready to submit a request for a raw socket descriptor \*/

if ((sd = socket (PF\_PACKET, SOCK\_RAW, htons (ETH\_P\_ALL))) < 0) {

perror ("socket() failed ");

exit (EXIT\_FAILURE);

}

There is still one critical piece of information that is missing here. When we generate the Ethernet packet, we need to add the destination MAC address of the local IPv6 gateway. One way is to use the *ifconfig* cli command and pull this information manually, assuming of course that such a command exists on your platform. We can also try to perform this automatically by listening for periodic IPv6 router advertisements on the interface. The MAC source address of these router advertisements packets is the destination address we are after. Here’s a code snippet that binds to the interface and looks for RA messages:

/\* Request a socket descriptor sd \*/

if ((sd = socket (AF\_INET6, SOCK\_RAW, IPPROTO\_ICMPV6)) < 0) {

perror ("Failed to get socket descriptor ");

exit (EXIT\_FAILURE);

}

/\* Set flag so we receive destination address from recvmsg \*/

on = 1;

if ((status = setsockopt (sd, IPPROTO\_IPV6, IPV6\_RECVPKTINFO, &on, sizeof (on))) < 0) {

perror ("setsockopt to IPV6\_RECVPKTINFO failed ");

exit (EXIT\_FAILURE);

}

/\* Obtain MAC address of this interface \*/

memset (&ifr, 0, sizeof (ifr));

snprintf (ifr.ifr\_name, sizeof (ifr.ifr\_name), "%s", interface);

if (ioctl (sd, SIOCGIFHWADDR, &ifr) < 0) {

perror ("ioctl() failed to get source MAC address ");

exit (EXIT\_FAILURE);

}

/\* Retrieve interface index of this node \*/

if ((ifindex = if\_nametoindex (interface)) == 0) {

perror ("if\_nametoindex() failed to obtain interface index ");

exit (EXIT\_FAILURE);

}

/\* Bind a device socket to this interface \*/

if (setsockopt (sd, SOL\_SOCKET, SO\_BINDTODEVICE, (void \*) &ifr, sizeof (ifr)) < 0) {

perror ("SO\_BINDTODEVICE failed");

exit (EXIT\_FAILURE);

}

/\* Listen for incoming message from socket sd

Keep at it until we get a router advertisement \*/

ra = (struct nd\_router\_advert \*) inpack;

while (ra->nd\_ra\_hdr.icmp6\_type != ND\_ROUTER\_ADVERT) {

if ((len = recvmsg (sd, &msghdr, 0)) < 0) {

perror ("recvmsg failed ");

exit (EXIT\_FAILURE);

}

}

pkt = (uint8\_t \*) inpack;

for (i=2; i<=7; i++) {

dst\_mac[i-2] = pkt[sizeof (struct nd\_router\_advert) + i];

}

close (sd);

We start the raw socket write process by generating a IPv6 UDP packet.

/\* IPv6 header \*/

iphdr = (struct ip6\_hdr \*) &out\_packet\_buffer[0] ;

/\* IPv6 version (4 bits), Traffic class (8 bits), Flow label (20 bits) \*/

iphdr->ip6\_flow = htonl ((6 << 28) | (0 << 20) | 0);

/\* Next header (8 bits): 44 for Frag \*/

iphdr->ip6\_nxt = 44;

/\* Hop limit (8 bits): default to maximum value \*/

iphdr->ip6\_hops = 255;

/\* src address \*/

bcopy(&srcaddr->sin6\_addr,&(iphdr->ip6\_src), 16) ;

/\* dst address \*/

bcopy(&cliaddr->sin6\_addr,&(iphdr->ip6\_dst), 16);

/\* set up the UDP packet \*/

uhdr = (struct udphdr \*) &(payload[0]);

uhdr->uh\_sport = htons(port) ;

uhdr->uh\_dport = cliaddr->sin6\_port;

uhdr->uh\_ulen = htons(dns\_response->len + 8);

uhdr->uh\_sum = 0;

/\* copy payload bytes from the dns response buffer to the payload buffer \*/

bcopy(dns\_response->buf,&payload[8],dns\_response->len) ;

/\* calculate the UDP checksum \*/

uhdr->uh\_sum = udp\_checksum(uhdr,dns\_response->len + 8, &srcaddr->sin6\_addr,

&cliaddr->sin6\_addr);

We have deliberately kept the IPv6 packet header and the UDP header and DNS response payload in separate memory buffers. This will allow us to place an IPv6 Extension Header into each output packet between the IPv6 packet header and each component part of the fragmented UDP header and payload, if we wish to fragment the output packet. In this case, the packet handler code will attempt to fragment any response greater than 16 octets in size.

/\* now fragment the output \*/

/\* set up the frag header \*/

fhdr = (struct ip6\_frag \*) &out\_packet\_buffer[40];

/\* the next header is a UDP packet header \*/

fhdr->ip6f\_nxt = 17 ;

fhdr->ip6f\_reserved = 0 ;

fhdr->ip6f\_offlg = htons(1);

fhdr->ip6f\_ident = rand() % 4294967296 ;

/\* the total size to send is the payload size plus the UDP header \*/

to\_send = dns\_response->len + 8 ;

to\_buf = (char \*) uhdr ;

/\* now carve up the UDP response into frags \*/

/\* initial block of UDP payload size is at least 8 bytes less than the original

dns response \*/

units = dns\_response->len / 8 ;

if (units > 16) datalen = 128 ;

else datalen = (units - 1) \* 8 ;

frag\_offset = 0 ;

/\* add in the size of the UDP header into datalen in the first instance \*/

datalen += 8 ;

/\* Destination and Source MAC addresses \*/

memcpy(ether\_frame, dst\_mac, 6 \* sizeof (uint8\_t));

memcpy(ether\_frame + 6, src\_mac, 6 \* sizeof (uint8\_t));

/\* Next is ethernet type code (ETH\_P\_IPV6 for IPv6) \*/

ether\_frame[12] = ETH\_P\_IPV6 / 256;

ether\_frame[13] = ETH\_P\_IPV6 % 256;

/\* now send the payload in fragments \*/

while (to\_send > 0) {

/\* each time we send datalen bytes plus the 8 byte frag header \*/

iphdr->ip6\_plen = htons(datalen + 8);

/\* now assemble the ether frame using 2 x 6 octet MAC addresses, a 2 octet

Ethernet frame type field, a 40 octet IPv6 header, a 8 octet Extension

Header and the payload \*/

frame\_length = 6 + 6 + 2 + 40 + 8 + datalen;

/\* IPv6 header + frag header \*/

memcpy (ether\_frame + ETH\_HDRLEN, iphdr, 48);

/\* payload fragment \*/

memcpy (ether\_frame + ETH\_HDRLEN + 48, to\_buf, datalen);

/\* Send ethernet frame out using the raw socket \*/

if ((bytes = sendto (sd, ether\_frame, frame\_length, 0,

(struct sockaddr \*) &device, sizeof (device))) <= 0) {

perror ("sendto() failed");

exit (EXIT\_FAILURE);

}

to\_send -= datalen ;

to\_buf += datalen ;

if (to\_send > 0) {

if (to\_send <= 512) {

/\* last frag \*/

frag\_offset += (datalen / 8) ;

fhdr->ip6f\_offlg = htons(frag\_offset << 3) ;

datalen = to\_send ;

}

else {

frag\_offset += (datalen / 8) ;

fhdr->ip6f\_offlg = htons((frag\_offset << 3) + 1);

datalen = 512 ;

}

}

}

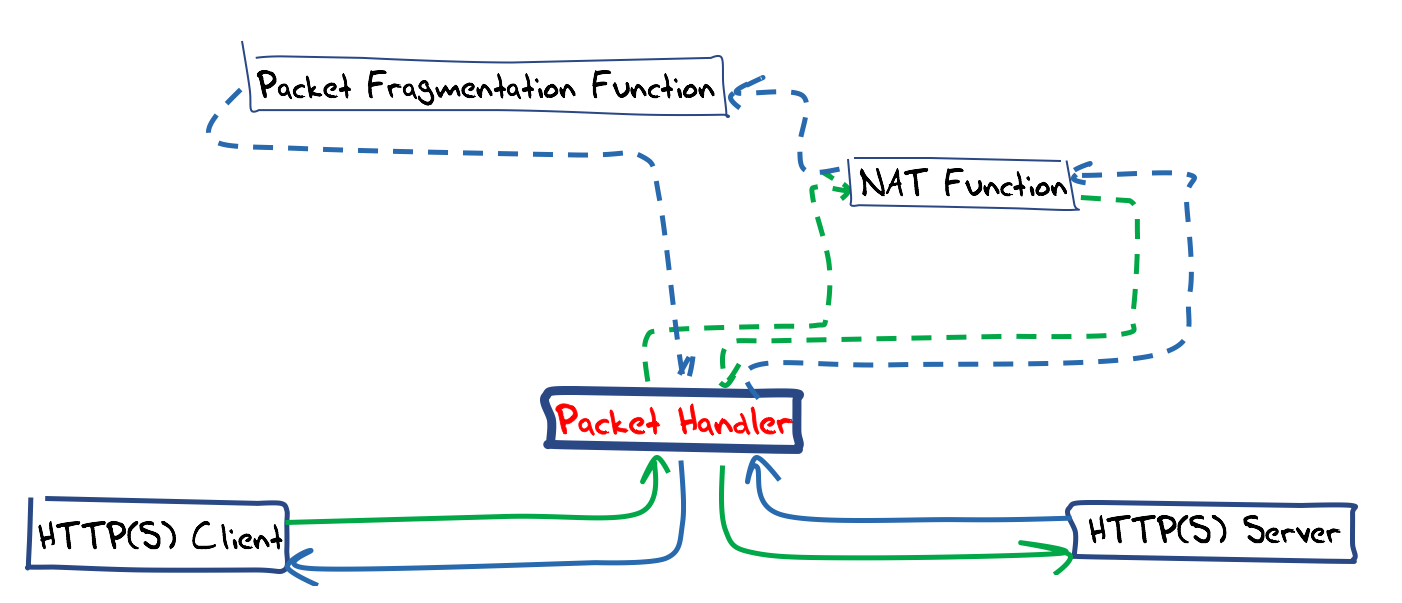
It should be noted that this is only a UDP interceptor, and if a client attempts to connect using TCP, then this packet handler will not be invoked A complete DNS front end would also perform TCP interception. However, this is not so straight forward as UDP, as we will see when we look at a fragmenting TCP packet handler in the next section.

The complete code for this UDP DNS packet handler can be found in a [GitHub repository](https://github.com/gih900/IPv6--DNS-Frag-Test-Rig).

A Raw Socket TCP HTTP(S) Server

The second part of our experiment was to measure the drop rate of fragmented IPv6 packets that were directed towards end users. Within the constraints of our experimental setup this implied that we need to place a packet fragmentation module into a TCP session. The approach taken here is similar to that used for UDP, namely using a ‘normal’ HTTP(S) server as a backend server, and implementing a packet handler to perform the fragmentation of outbound packets directed to the end client.

There are a number of possible implementation options. The chosen option was to use a stateless packet handler along the lines of an IPv6 NAT. Incoming SYN packets from the client create a new NAT binding state and the packet headers are re-written so that the packet destination address is that of the back end HTTP processor and the packet source address is this host. Other incoming packets from the client were matched against an established NAT binding state, the packet headers were re-written and passed to the back end. Packets from the back end needed to be matched against an established NAT state, and the packet headers were translated and large payload responses were fragmented as they were passed to the client. This design is shown in Figure 2.

*[](http://www.potaroo.net/ispcol/2017-10/raw-fig2.png)  
Figure 2 – TCP Packet Handler*

The first task is to receive TCP packets from the client and get them passed to the application, bypassing the operating system’s conventional behaviour. Most Unix systems will recognise an incoming TCP packet and if there is no listening TCP port associated with the destination port number, the system will respond with a TCP reset (RST) packet. If there is a TCP session associated with the port, then the operating system’s IP and TCP drivers will process the packet and our raw packet handlers will not be invoked in any case. We need to stop this automatic RST generation when receiving unbound TCP packets.

In FreeBSD systems there is a system parameter than can be set to turn off this behaviour:

sysctl -w net.tcp.blackhole=2.

Debian Linux systems don’t appear to have this form of control over these kernel-generated TCP reset responses, so the alternative is to suppress the TCP reset packet before it leaves the system. The approach we used was to set up a filter in outbound packets using an iptables filter entry:

ip6tables -I OUTPUT 1 -o eth0 -p tcp --tcp-flags ALL RST -j DROP

Strictly speaking this does not suppress the generation of the RST packet, but stops it leaving the system, which is the same overall result as suppression in any case.

We then need to pick up all incoming IPv6 TCP packets addressed to the listening address on port 80 and port 443 and process them within the context of this application. The simplest way to achieve this is by using the packet capture library routines (*libpcap*).

/\* the PCAP capture filter - http and https v6 traffic only\*/

sprintf(filter\_exp,"dst host %s and (port 80 or port 443) and tcp and ip6",host);

/\* open capture device \*/

if ((handle = pcap\_open\_live(interface, SNAP\_LEN, 1, 1, errbuff)) == NULL) {

fprintf(stderr, "Couldn't open device %s: %s\n", interface, errbuff);

exit(EXIT\_FAILURE) ;

}

/\* compile the filter expression \*/

if (pcap\_compile(handle, &fp, filter\_exp, 0, 0) == -1) {

fprintf(stderr, "Couldn't parse filter %s: %s\n", filter\_exp, pcap\_geterr(handle));

exit(EXIT\_FAILURE) ;

}

/\* install the filter \*/

if (pcap\_setfilter(handle, &fp) == -1) {

fprintf(stderr, "Couldn't install filter %s: %s\n", filter\_exp, pcap\_geterr(handle));

exit(EXIT\_FAILURE) ;

}

/\* set up the packet capture in an infinite loop \*/

if (debug) printf("Enter PCAP packet capture loop\n") ;

pcap\_loop(handle, -1, got\_packet, NULL) ;

Each incoming packet will invoke the packet handler got\_packet(), which can be used to process both packets coming in from the client and packets coming in from the HTTP(S) back end.

However, before looking at this code there is one more aspect of many modern host systems that we need to disable. These days many Ethernet interfaces are “smart” and part of that additional functionality is to perform TCP segmentation offloading into the interface card. The argument in favour of this is one of relieving the kernel of extraneous I/O interrupts, allowing the kernel to pass large buffers of memory to the interface card and have the card perform TCP segmentation for outgoing packets. Similarly, the card may aggregate several arriving packets into a single pseudo-TCP segment that is passed to the operating system in a single interrupt transaction. Obviously, we need to disable this functionality in this context. The ethtool package provides a way to manage these I/O devices:

ethtool –K eth0 generic-segmentation-offload off

ethtool –K eth0 generic-receive-offload off

We can now examine the major aspects of the packet handler within the got\_packet() routine. The first is the handling of packets that come from the end client. The source address and source port are used to lookup the local NAT translation table, and this is used to create an outbound packet destined to the back-end HTTP(S) server.

/\* this is a packet from the client to the packet handler \*/

if ((dport == 80) || (dport === 443)) {

/\* search of a matching entry in the local NAT table using the

source address and source port as the lookup key \*/

bcopy(&(ip->ip6\_src), &(bdg.ip6\_src),16);

bdg.sport = sport ;

bdp = find\_binding(&bdg,dport) ;

/\* if we cannot find a NAT table entry, and the packet contains a TCP

SYN flag, then we can create a new NAT entry \*/

if ((!bdp) && (tcp->th\_flags & TH\_SYN)) {

if (dport == 80) {

port = next\_port\_80(&(ip->ip6\_src),ntohs(tcp->th\_sport)) ;

bdp = port\_80\_ptr[port]->entry;

}

else {

port = next\_port\_443(&(ip->ip6\_src),ntohs(tcp->th\_sport)) ;

bdp = port\_443\_ptr[port]->entry;

}

}

else if (!bdp)

return;

/\* retain the current TCP session sequence number in the NAT table \*/

bdp->seq = tcp->th\_seq ;

/\* perform a header translation and pass the packet to the back-end

http(s) server \*/

send\_packet\_to\_http(packet,bdp) ;

}

The packets received from the back end have the ‘reverse’ NAT header substitution applied in both the IPv6 and TCP packet headers. The TCP checksum on the TCP pseudo-header is computed and the response is ready for the final phase of sending.

/\* IPv6 header \*/

iphdr = (struct ip6\_hdr \*) &out\_packet\_buffer[0] ;

o\_iphdr = (struct ip6\_hdr \*) (packet + ETH\_HDRLEN) ;

/\* IPv6 version (4 bits), Traffic class (8 bits), Flow label (20 bits) \*/

iphdr->ip6\_flow = o\_iphdr->ip6\_flow ;

/\* payload length \*/

len = ntohs(o\_iphdr->ip6\_plen);

/\* Hop limit (8 bits): default to maximum value \*/

iphdr->ip6\_hops = 255;

/\* src address \*/

bcopy(&local6\_addr,&(iphdr->ip6\_src), 16) ;

/\* dst address \*/

bcopy(&tp->ip6\_src,&(iphdr->ip6\_dst), 16);

/\* TCP header \*/

orig\_tcp = (struct tcphdr \*) (packet + ETH\_HDRLEN + 40);

tcp = (struct tcphdr \*) &(out\_packet\_buffer[40]);

tcp->th\_dport = htons(tp->sport) ;

tcp->th\_sport = orig\_tcp->th\_sport;

/\* copy payload bytes from the original packet to the payload buffer \*/

memcpy(&out\_packet\_buffer[44],&packet[ETH\_HDRLEN + 44],len - 4) ;

/\* Destination and Source MAC addresses \*/

memcpy(ether\_frame, dst\_mac, 6 \* sizeof (uint8\_t));

memcpy(ether\_frame + 6, src\_mac, 6 \* sizeof (uint8\_t));

/\* Next is ethernet type code (ETH\_P\_IPV6 for IPv6) \*/

ether\_frame[12] = ETH\_P\_IPV6 / 256;

ether\_frame[13] = ETH\_P\_IPV6 % 256;

/\* Copy the IPv6 header into the ether\_frame \*/

memcpy(ether\_frame + ETH\_HDRLEN, &out\_packet\_buffer[0], 40);

payload = ntohs(iphdr->ip6\_plen) - (tcp->th\_off \* 4) ;

pptr = &out\_packet\_buffer[40 + (tcp->th\_off \* 4)];

tcp\_hdr\_len = tcp->th\_off \* 4 ;

tcp\_sequence = ntohl(tcp->th\_seq) ;

If the payload is small we’ll send it as is, without additional fragmentation.

if (payload <= 16) {

/\* copy across the TCP header \*/

memcpy(ether\_frame + ETH\_HDRLEN + 40, &out\_packet\_buffer[40], tcp\_hdr\_len);

/\* copy across the payload \*/

memcpy(ether\_frame + ETH\_HDRLEN + 40 + tcp\_hdr\_len, pptr, payload);

/\* IPv6 payload length \*/

e\_iphdr = (struct ip6\_hdr \*) (ether\_frame + ETH\_HDRLEN) ;

e\_iphdr->ip6\_plen = htons(tcp\_hdr\_len + payload);

/\* Next header (8 bits): 6 for TCP \*/

e\_iphdr->ip6\_nxt = IPPROTO\_TCP;

/\* put in the adjusted sequence number and the new checksum \*/

e\_tcp = (struct tcphdr \*) (ether\_frame + ETH\_HDRLEN + 40) ;

e\_tcp->th\_seq = htonl(tcp\_sequence) ;

e\_tcp->th\_sum = 0 ;

e\_tcp->th\_sum = tcp\_checksum(e\_tcp,tcp\_hdr\_len+payload,tcp\_hdr\_len+payload,

&(e\_iphdr->ip6\_src),&(e\_iphdr->ip6\_dst)) ;

/\* ethernet frame length \*/

frame\_length = ETH\_HDRLEN + 40 + tcp\_hdr\_len + payload ;

if ((bytes = sendto (sd, ether\_frame, frame\_length, 0,

(struct sockaddr \*) &device, sizeof (device))) <= 0) {

perror ("sendto() failed");

exit (EXIT\_FAILURE);

}

return;

}

We are using the raw socket *sd* to send this packet. Given that we are already using the *pcap* library to receive these packets directly off the interface an alternative would be to use the pcap\_sendpacket() routine instead. This alternative approach would be more portable across operating system platforms, as it would no longer be reliant on a particular form of interface control functions that are only exposed in Linux systems, but perhaps this alternative approach is best left as an exercise for an enthusiastic reader.

For larger payloads we’ll perform re-segmentation and fragmentation. We’ll re-segment each packet so that each TCP packet it is no larger than 1,248 octets, and then we’ll fragment the packet into two parts, with the trailing frag containing the last 256 octets of the payload.

while (payload > 0) {

/\* lets pull off the trailing 8 (or so bytes) into a fragmented trailer \*/

if (payload < 1200) {

tcp\_seg\_len = payload ;

this\_frag = ((payload / 8) - 1) \* 8 ;

}

/\* lets re-segment the packet \*/

if (payload >= 1200) {

tcp\_seg\_len = 1200 ;

this\_frag = 1200 - 256 ;

}

/\* start constructing the IPv6 header in the Ethernet frame

by writing in the IPv6 payload length and next header fields \*/

e\_iphdr = (struct ip6\_hdr \*) (ether\_frame + ETH\_HDRLEN) ;

e\_iphdr->ip6\_plen = htons(8 + tcp\_hdr\_len + this\_frag);

e\_iphdr->ip6\_nxt = 44 ; // Fragmentation header

/\* now set up the frag header \*/

fhdr = (struct ip6\_frag \*) ðer\_frame[ETH\_HDRLEN+40] ;

fhdr->ip6f\_nxt = IPPROTO\_TCP ;

fhdr->ip6f\_reserved = 0 ;

fhdr->ip6f\_offlg = htons(1); // Offset is zero and Set more-fragments flag

fhdr->ip6f\_ident = rand() % 4294967296 ;

/\* for the first frag, copy across the TCP header \*/

memcpy(ether\_frame + ETH\_HDRLEN + 40 + 8, &out\_packet\_buffer[40], tcp\_hdr\_len);

/\* copy across the entire payload (in order to generate the correct tcp checksum) \*/

memcpy(ether\_frame + ETH\_HDRLEN + 40 + 8 + tcp\_hdr\_len, pptr, tcp\_seg\_len);

/\* put in the adjusted sequence number and the new checksum \*/

e\_tcp = (struct tcphdr \*) (ether\_frame + ETH\_HDRLEN + 40 + 8) ;

e\_tcp->th\_seq = htonl(tcp\_sequence) ;

e\_tcp->th\_sum = 0 ;

e\_tcp->th\_sum = tcp\_checksum(e\_tcp,tcp\_hdr\_len + tcp\_seg\_len,

tcp\_hdr\_len + tcp\_seg\_len,

&(e\_iphdr->ip6\_src),&(e\_iphdr->ip6\_dst)) ;

/\* send the leading fragment to the Ethernet interface \*/

frame\_length = ETH\_HDRLEN + 40 + 8 + tcp\_hdr\_len + this\_frag ;

if ((bytes = sendto (sd, ether\_frame, frame\_length, 0, (struct sockaddr \*) &device,

sizeof (device))) <= 0) {

perror ("sendto() failed");

exit (EXIT\_FAILURE);

}

pptr += this\_frag;

offset = (this\_frag + tcp\_hdr\_len) >> 3;

remainder = tcp\_seg\_len - this\_frag ;

/\* now adjust the frag header for the trailing frag \*/

fhdr->ip6f\_offlg = htons(offset << 3); // Offset

this\_frag = remainder ;

e\_iphdr->ip6\_plen = htons(8 + this\_frag);

/\* copy across the remainder of the payload immediately folloing the frag header \*/

memcpy(ether\_frame + ETH\_HDRLEN + 40 + 8, pptr, this\_frag);

/\* send the trailing fragment \*/

frame\_length = ETH\_HDRLEN + 40 + 8 + this\_frag ;

if ((bytes = sendto (sd, ether\_frame, frame\_length, 0, (struct sockaddr \*) &device,

sizeof (device))) <= 0) {

perror ("sendto() failed");

exit (EXIT\_FAILURE);

}

pptr += this\_frag ;

tcp\_sequence += tcp\_seg\_len ;

payload -= tcp\_seg\_len ;

}

The complete code for this TCP packet handler can be found in a [GitHub repository](https://github.com/gih900/IPv6--TCP-Frag-Test-Rig).

I find it to be a little odd that it’s so challenging to set up a raw socket interface in IPv6. It seems that the challenges start with interface cards that by default want to perform TCP segmentation and UDP fragmentation on output and perform the reverse function on input. Then we have some operating systems that seem to be hardwired to send reset packets when receiving TCP packets that to not address an open TCP socket. Then we have the IPv6 protocol engine that simply does not support raw socket connections in the same manner as IPv4.

The good news, however, is that an equivalent level of functionality can be coerced to work on most systems, and even where it does not appear that the ethernet level socket calls are supported, the *libpcap* library can be used in its place, as this library supports both sending and receiving packets.

**🎯 Introduction**

In the realm of network programming, raw sockets provide a powerful tool for developers to access and manipulate network packets at a low level. They allow the programmer to work directly with the network layer, bypassing the transport layer (e.g., TCP, UDP). This level of access is particularly useful for crafting custom network protocols, network monitoring, and network security applications. In this blog post, we will delve into the concept of raw sockets, provide a detailed explanation of the C program, and explore its output to understand how it reads and interprets raw network packets.

**🎯 Concept of RAW Sockets**

A raw socket is a type of socket that provides direct access to the underlying network protocols, such as IP (Internet Protocol). It enables the programmer to handle the IP header and data directly, giving full control over the entire packet, including its headers and payloads. Unlike higher-level sockets, like SOCK\_STREAM (TCP) and SOCK\_DGRAM (UDP), which handle data at the transport layer, raw sockets work at the network layer.

Using raw sockets, developers can build custom protocols, implement low-level network monitoring and sniffing tools, perform network diagnostics, and even create custom firewalls or intrusion detection systems. However, working with raw sockets requires a good understanding of network protocols and imposes additional responsibilities on the developer, such as checksum calculation and handling protocol-specific intricacies.

**🎯 Detailed Explanation of the C Program**

**👉 Source Code**

#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/ip.h>

int main() {

    int sockfd;

    char buff[8192] = "";

    struct iphdr \*ip;

    sockfd = socket(AF\_INET, SOCK\_RAW, IPPROTO\_ICMP);

    while (read(sockfd, buff, 8192)) {

        ip = (struct iphdr \*)buff;

        printf("\nSource: %s \n", inet\_ntoa(ip->saddr));

        printf("Destination: %s\n", inet\_ntoa(ip->daddr));

    }

    return 0;

}

**👉 Explanation**

In this C program, we use a raw socket to receive and read network packets. Let's break down the important parts of the code:

**Header Files:**

The necessary header files are included to access socket-related functions and structures.

**Socket Creation:**

The socket() function is called to create a raw socket. The arguments passed are AF\_INET (IPv4), SOCK\_RAW (raw socket type), and IPPROTO\_ICMP (the protocol we want to capture, in this case, ICMP). Raw sockets allow us to receive all packets of the specified protocol.

**Packet Reception:**

The program enters a while loop to continuously read packets from the raw socket using the read() system call. The function reads up to 8192 bytes of data from the socket and stores it in the buff buffer.

**Packet Analysis:**

The received packet is then treated as an IP packet. We cast the buffer buff to a pointer of type struct iphdr (IP header structure) named ip. This allows us to access the IP header fields directly.

**Source and Destination Printing:**

We extract the source and destination IP addresses from the IP header using inet\_ntoa, a function that converts IP addresses from binary to a readable string format. The extracted addresses are then printed to the console.

**🎯 Sample Output**

[root@localhost ~]# ./a.out

Source: 172.16.29.123

Destnation: 172.16.29.60

Source: 172.16.29.123

Destnation: 172.16.29.60

Source: 172.16.29.123

Destnation: 172.16.29.60

Source: 172.16.29.123

Destnation: 172.16.29.60

Source: 172.16.29.123

Destnation: 172.16.29.60

Source: 172.16.29.123

Destnation: 172.16.29.60

Source: 172.16.29.123

Destnation: 172.16.29.60

**🎯 Summary**

In this blog post, we explored the concept of raw sockets in Unix using C programming. We learned that raw sockets provide direct access to network protocols, allowing developers to handle IP packets at a low level. The C program provided reads ICMP packets from a raw socket and prints the source and destination IP addresses from each packet.

**🎯 Key Points**

* Raw sockets offer direct access to network protocols at the network layer.
* They are powerful tools for building custom protocols, network monitoring, and network security applications.
* The C program demonstrated how to create and use a raw socket to read ICMP packets and extract IP addresses.

Layer 2 Raw Socket Programming

We assume [the environment](https://iplab.naist.jp/class/2018/materials/hands-on/vm-preparation/) introduced in the previous lecture. In this hands-on session, we will create data link raw sockets using Linux and Python 3 and try to send/receive data using sockets.

Hands-On with Sample Code

Step 1. Preparation

[Here](https://github.com/y-sira/pyng) is the sample code. (This session is based on this Python sample code. If you prefer C language, please refer to [the C version](https://github.com/shun-yo/cethping).)

Open the terminal and type:

$ git clone https://github.com/y-sira/pyng.git

You can find network interface names and MAC addresses on your virtual machine using ip command as follows:

$ ip link

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc fq\_codel state UP mode DEFAULT group default qlen 1000

link/ether 08:00:27:dd:d7:43 brd ff:ff:ff:ff:ff:ff

3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc fq\_codel state UP mode DEFAULT group default qlen 1000

link/ether 08:00:27:b0:d6:ff brd ff:ff:ff:ff:ff:ff

4: enp0s9: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc fq\_codel state UP mode DEFAULT group default qlen 1000

link/ether 08:00:27:e6:4d:39 brd ff:ff:ff:ff:ff:ff

5: enp0s10: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc fq\_codel state UP mode DEFAULT group default qlen 1000

link/ether 08:00:27:8e:75:44 brd ff:ff:ff:ff:ff:ff

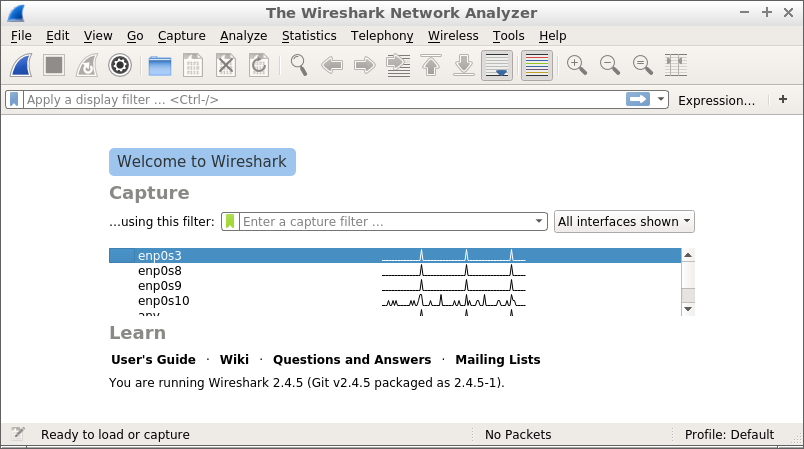
Please remember interface names and corresponding MAC addresses.

Step 2. Start Packet Capture Using Wireshark

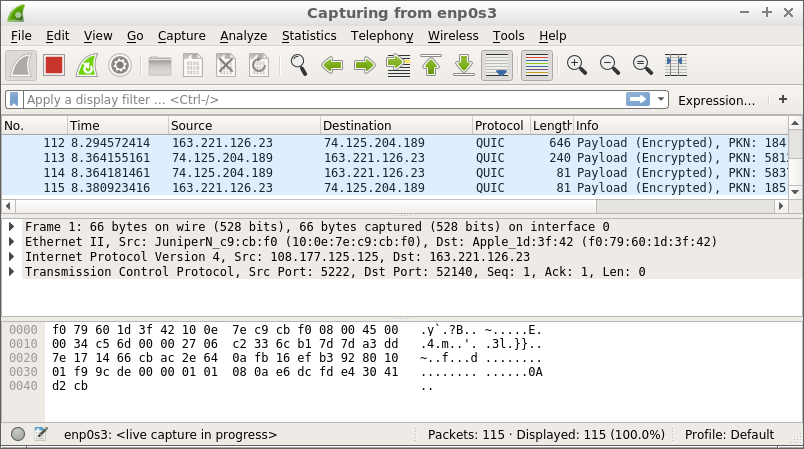
Open the terminal and type:

$ sudo wireshark

Wireshark screen will be appeared.



Double click the interface name you want to monitor, then you can see packets sent to the interface.



Step 3. Send/Receive Frames Using Layer 2 Raw Sockets

Move to pyng directory and run the server script:

$ cd /path/to/pyng

$ sudo ./pyngd $SERVER\_INTERFACE\_NAME

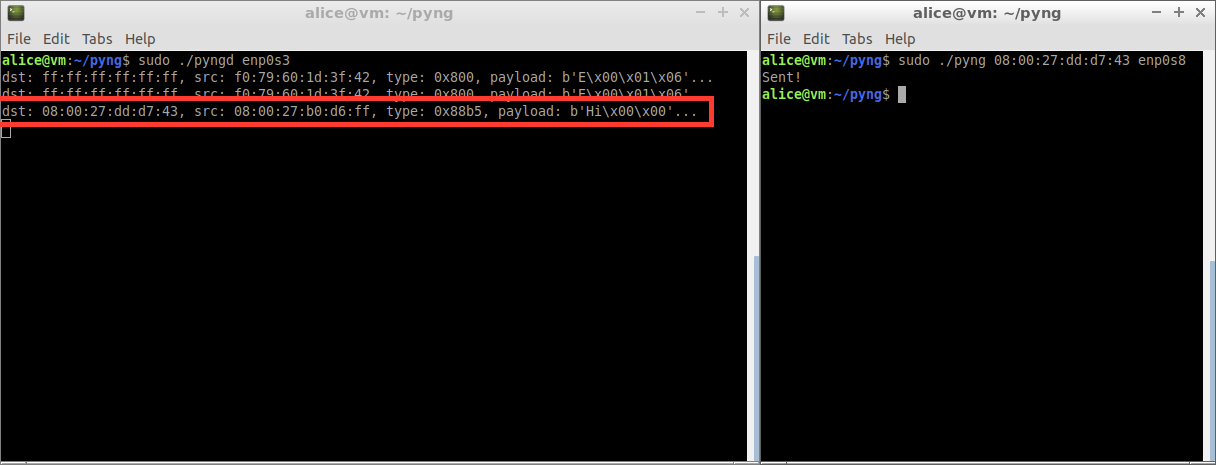
Open another terminal, move to pyng directory, and run the client script to send the frame to the server:

$ cd /path/to/pyng

$ sudo ./pyng $DESTINATION\_MAC\_ADDRESS $CLIENT\_INTERFACE\_NAME

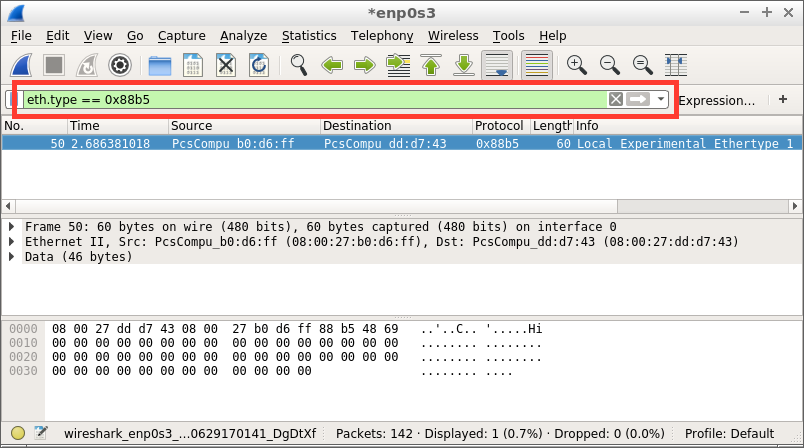
Sent!

Make sure that the data client script sent is appeared in the server side terminal.



Step 4. Check Frames with Wireshark

Stop packet capture. You can filter the packet by protocol, source address, destination address, …, etc. Please enter eth.type == 0x88b5 in the filter text box to check frames you sent.



Verify the Ethernet header and the payload.

Step 5. Considerations

Please consider below.

* What data is sent?
  + Why does the payload contain many zero?
* How do we send/receive data in the sample code?
  + What is struct.pack('!6s6sH', ...) in the server side script?
* What does the magic number 0x88b5 mean?
* What kinds of packets are displayed in Wireshark?
* What does the MAC address ff:ff:ff:ff:ff:ff mean?
* What is select in the server side sample code?
  + Why do not use recv() directly?
  + How can we handle massive requests?
* …

For Beginners

Socket APIs

The socket APIs are APIs for interprocess communication and network communication from applications. It is available on UNIX and Linux. In Linux, it is easy to handle the data link layer by using the socket API. Let’s explain Linux sockets later.

Data Link Raw Sockets

In order to send and receive data using socket APIs, we have to generate a socket descriptor. The socket descriptor can be generated using int socket (int family, int type, int protocol) defined in sys/socket.h. The first argument family specifies the address family. For the data link layer, specify AF\_PACKET defined in sys/socket.h. The second argument type specifies the socket type. For the data link layer, specify SOCK\_RAW defined in sys/socket.h. The third argument protocol specifies the protocol. Specify ETH\_P\_ALL to retrieve any Ethernet frames, and specify ETH\_P\_IP to retrieve Ethernet frames containing a IP packet. The protocol types of the Ethernet frame is defined in linux/if\_ether.h.

In Python, there is a standard library that wraps the OS level socket APIs, so we will use it to create a data link raw socket. Please be careful to close() after using the socket.

import socket

ETH\_P\_ALL **=** 3

s **=** socket**.**socket(socket**.**AF\_PACKET, socket**.**SOCK\_RAW, socket**.**htons(ETH\_P\_ALL))

s**.**close()

We use bind() to bind the network interface to the socket. You can check the network interface available on your computer with the following command.

$ ip link

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc fq\_codel state UP mode DEFAULT group default qlen 1000

link/ether 08:00:27:dd:d7:43 brd ff:ff:ff:ff:ff:ff

3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc fq\_codel state UP mode DEFAULT group default qlen 1000

link/ether 08:00:27:b0:d6:ff brd ff:ff:ff:ff:ff:ff

4: enp0s9: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc fq\_codel state UP mode DEFAULT group default qlen 1000

link/ether 08:00:27:e6:4d:39 brd ff:ff:ff:ff:ff:ff

5: enp0s10: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc fq\_codel state UP mode DEFAULT group default qlen 1000

link/ether 08:00:27:8e:75:44 brd ff:ff:ff:ff:ff:ff

For example, to bind the network interface enp0s3 to the socket, write as follows.

import socket

ETH\_P\_ALL **=** 3

interface **=** 'enp0s3'

s **=** socket**.**socket(socket**.**AF\_PACKET, socket**.**SOCK\_RAW, socket**.**htons(ETH\_P\_ALL))

s**.**bind((interface, 0))

*# do something*

s**.**close()

Now you are ready to send and receive raw data on the data link layer using the network interface enp0s3.

Receive Data Using Raw Sockets

Use recv() to receive data using a socket. Specify the buffer size in first argument of recv(). recv() will block until ready to receive data from the socket. The return value of recv() is the received byte sequence. This byte string consists of the header and the payload of the Ethernet frame.

When receiving data sent to enp0s3, the script server.py is as follows:

import socket

ETH\_P\_ALL **=** 3

ETH\_FRAME\_LEN **=** 1514 *# Max. octets in frame sans FCS*

interface **=** 'enp0s3'

s **=** socket**.**socket(socket**.**AF\_PACKET, socket**.**SOCK\_RAW, socket**.**htons(ETH\_P\_ALL))

s**.**bind((interface, 0))

data **=** s**.**recv(ETH\_FRAME\_LEN)

**print**(data) *# => b'\x08\x00\x27\xdd\xd7\x43\x08\x00\x27\x8e\x75\x44\x88\xb5Hi'*

s**.**close()

Note that the data link raw sockets require a root permission. Please run the above script as root.

When the data is sent to enp0s3, you can get the following output:

$ sudo python3 server.py

b"\x08\x00'\xdd\xd7C\x08\x00'\x8euD\x88\xb5Hi"

Send Data Using Raw Sockets

To send data using a socket, use sendall(). The data to be sent must be a byte sequence containing the header and payload of the Ethernet frame.

When sending data “Hi” from enp0s10 (08:00:27:8e:75:44) to enp0s3 (08:00:27:dd:d7:43), the script client.py is as follows:

import socket

ETH\_P\_ALL **=** 3

interface **=** 'enp0s10'

dst **=** b'\x08\x00\x27\xdd\xd7\x43' *# destination MAC address*

src **=** b'\x08\x00\x27\x8e\x75\x44' *# source MAC address*

proto **=** b'\x88\xb5' *# ethernet frame type*

payload **=** 'Hi'**.**encode() *# payload*

s **=** socket**.**socket(socket**.**AF\_PACKET, socket**.**SOCK\_RAW, socket**.**htons(ETH\_P\_ALL))

s**.**bind((interface, 0))

s**.**sendall(dst **+** src **+** proto **+** payload)

s**.**close()

Then,

$ sudo python3 client.py

Sent!

Make sure that the server side terminal outputs the data you sent.

socket — Low-level networking interface

**Source code:** [Lib/socket.py](https://github.com/python/cpython/tree/3.13/Lib/socket.py)

This module provides access to the BSD *socket* interface. It is available on all modern Unix systems, Windows, MacOS, and probably additional platforms.

**Note**

Some behavior may be platform dependent, since calls are made to the operating system socket APIs.

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

This module does not work or is not available on WebAssembly. See [WebAssembly platforms](https://docs.python.org/3/library/intro.html" \l "wasm-availability) for more information.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python’s object-oriented style: the [socket()](https://docs.python.org/3/library/socket.html#socket.socket) function returns a *socket object* whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with read() and write() operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

**See also**

**Module**[socketserver](https://docs.python.org/3/library/socketserver.html" \l "module-socketserver" \o "socketserver: A framework for network servers.)

Classes that simplify writing network servers.

**Module**[ssl](https://docs.python.org/3/library/ssl.html" \l "module-ssl" \o "ssl: TLS/SSL wrapper for socket objects)

A TLS/SSL wrapper for socket objects.

Socket families

Depending on the system and the build options, various socket families are supported by this module.

The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created. Socket addresses are represented as follows:

* The address of an [AF\_UNIX](https://docs.python.org/3/library/socket.html#socket.AF_UNIX) socket bound to a file system node is represented as a string, using the file system encoding and the 'surrogateescape' error handler (see [**PEP 383**](https://peps.python.org/pep-0383/)). An address in Linux’s abstract namespace is returned as a [bytes-like object](https://docs.python.org/3/glossary.html#term-bytes-like-object) with an initial null byte; note that sockets in this namespace can communicate with normal file system sockets, so programs intended to run on Linux may need to deal with both types of address. A string or bytes-like object can be used for either type of address when passing it as an argument.

*Changed in version 3.3:*Previously, [AF\_UNIX](https://docs.python.org/3/library/socket.html#socket.AF_UNIX) socket paths were assumed to use UTF-8 encoding.

*Changed in version 3.5:*Writable [bytes-like object](https://docs.python.org/3/glossary.html#term-bytes-like-object) is now accepted.

* A pair (host, port) is used for the [AF\_INET](https://docs.python.org/3/library/socket.html#socket.AF_INET) address family, where *host* is a string representing either a hostname in internet domain notation like 'daring.cwi.nl' or an IPv4 address like '100.50.200.5', and *port* is an integer.
  + For IPv4 addresses, two special forms are accepted instead of a host address: '' represents INADDR\_ANY, which is used to bind to all interfaces, and the string '<broadcast>' represents INADDR\_BROADCAST. This behavior is not compatible with IPv6, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.
* For [AF\_INET6](https://docs.python.org/3/library/socket.html#socket.AF_INET6) address family, a four-tuple (host, port, flowinfo, scope\_id) is used, where *flowinfo* and *scope\_id* represent the sin6\_flowinfo and sin6\_scope\_id members in struct sockaddr\_in6 in C. For [socket](https://docs.python.org/3/library/socket.html#module-socket) module methods, *flowinfo* and *scope\_id* can be omitted just for backward compatibility. Note, however, omission of *scope\_id* can cause problems in manipulating scoped IPv6 addresses.

*Changed in version 3.7:*For multicast addresses (with *scope\_id* meaningful) *address* may not contain %scope\_id (or zone id) part. This information is superfluous and may be safely omitted (recommended).

* AF\_NETLINK sockets are represented as pairs (pid, groups).
* Linux-only support for TIPC is available using the AF\_TIPC address family. TIPC is an open, non-IP based networked protocol designed for use in clustered computer environments. Addresses are represented by a tuple, and the fields depend on the address type. The general tuple form is (addr\_type, v1, v2, v3 [, scope]), where:
  + *addr\_type* is one of TIPC\_ADDR\_NAMESEQ, TIPC\_ADDR\_NAME, or TIPC\_ADDR\_ID.
  + *scope* is one of TIPC\_ZONE\_SCOPE, TIPC\_CLUSTER\_SCOPE, and TIPC\_NODE\_SCOPE.
  + If *addr\_type* is TIPC\_ADDR\_NAME, then *v1* is the server type, *v2* is the port identifier, and *v3* should be 0.

If *addr\_type* is TIPC\_ADDR\_NAMESEQ, then *v1* is the server type, *v2* is the lower port number, and *v3* is the upper port number.

If *addr\_type* is TIPC\_ADDR\_ID, then *v1* is the node, *v2* is the reference, and *v3* should be set to 0.

* A tuple (interface, ) is used for the [AF\_CAN](https://docs.python.org/3/library/socket.html#socket.AF_CAN) address family, where *interface* is a string representing a network interface name like 'can0'. The network interface name '' can be used to receive packets from all network interfaces of this family.
  + [CAN\_ISOTP](https://docs.python.org/3/library/socket.html#socket.CAN_ISOTP) protocol require a tuple (interface, rx\_addr, tx\_addr) where both additional parameters are unsigned long integer that represent a CAN identifier (standard or extended).
  + [CAN\_J1939](https://docs.python.org/3/library/socket.html#socket.CAN_J1939) protocol require a tuple (interface, name, pgn, addr) where additional parameters are 64-bit unsigned integer representing the ECU name, a 32-bit unsigned integer representing the Parameter Group Number (PGN), and an 8-bit integer representing the address.
* A string or a tuple (id, unit) is used for the SYSPROTO\_CONTROL protocol of the PF\_SYSTEM family. The string is the name of a kernel control using a dynamically assigned ID. The tuple can be used if ID and unit number of the kernel control are known or if a registered ID is used.

*Added in version 3.3.*

* AF\_BLUETOOTH supports the following protocols and address formats:
  + BTPROTO\_L2CAP accepts (bdaddr, psm) where bdaddr is the Bluetooth address as a string and psm is an integer.
  + BTPROTO\_RFCOMM accepts (bdaddr, channel) where bdaddr is the Bluetooth address as a string and channel is an integer.
  + BTPROTO\_HCI accepts a format that depends on your OS.
    - On Linux it accepts a tuple (device\_id,) where device\_id is an integer specifying the number of the Bluetooth device.
    - On FreeBSD, NetBSD and DragonFly BSD it accepts bdaddr where bdaddr is the Bluetooth address as a string.

*Changed in version 3.2:*NetBSD and DragonFlyBSD support added.

*Changed in version 3.13.3:*FreeBSD support added.

* + BTPROTO\_SCO accepts bdaddr where bdaddr is the Bluetooth address as a string or a [bytes](https://docs.python.org/3/library/stdtypes.html#bytes) object. (ex. '12:23:34:45:56:67' or b'12:23:34:45:56:67') This protocol is not supported under FreeBSD.
* [AF\_ALG](https://docs.python.org/3/library/socket.html#socket.AF_ALG) is a Linux-only socket based interface to Kernel cryptography. An algorithm socket is configured with a tuple of two to four elements (type, name [, feat [, mask]]), where:
  + *type* is the algorithm type as string, e.g. aead, hash, skcipher or rng.
  + *name* is the algorithm name and operation mode as string, e.g. sha256, hmac(sha256), cbc(aes) or drbg\_nopr\_ctr\_aes256.
  + *feat* and *mask* are unsigned 32bit integers.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 2.6.38.

Some algorithm types require more recent Kernels.

*Added in version 3.6.*

* [AF\_VSOCK](https://docs.python.org/3/library/socket.html#socket.AF_VSOCK) allows communication between virtual machines and their hosts. The sockets are represented as a (CID, port) tuple where the context ID or CID and port are integers.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 3.9

See *[vsock(7)](https://manpages.debian.org/vsock(7))*

*Added in version 3.7.*

* [AF\_PACKET](https://docs.python.org/3/library/socket.html#socket.AF_PACKET) is a low-level interface directly to network devices. The addresses are represented by the tuple (ifname, proto[, pkttype[, hatype[, addr]]]) where:
  + *ifname* - String specifying the device name.
  + *proto* - The Ethernet protocol number. May be [ETH\_P\_ALL](https://docs.python.org/3/library/socket.html#socket.ETH_P_ALL) to capture all protocols, one of the [ETHERTYPE\_\* constants](https://docs.python.org/3/library/socket.html#socket-ethernet-types) or any other Ethernet protocol number.
  + *pkttype* - Optional integer specifying the packet type:
    - PACKET\_HOST (the default) - Packet addressed to the local host.
    - PACKET\_BROADCAST - Physical-layer broadcast packet.
    - PACKET\_MULTICAST - Packet sent to a physical-layer multicast address.
    - PACKET\_OTHERHOST - Packet to some other host that has been caught by a device driver in promiscuous mode.
    - PACKET\_OUTGOING - Packet originating from the local host that is looped back to a packet socket.
  + *hatype* - Optional integer specifying the ARP hardware address type.
  + *addr* - Optional bytes-like object specifying the hardware physical address, whose interpretation depends on the device.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 2.2.

* [AF\_QIPCRTR](https://docs.python.org/3/library/socket.html#socket.AF_QIPCRTR) is a Linux-only socket based interface for communicating with services running on co-processors in Qualcomm platforms. The address family is represented as a (node, port) tuple where the *node* and *port* are non-negative integers.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 4.7.

*Added in version 3.8.*

* IPPROTO\_UDPLITE is a variant of UDP which allows you to specify what portion of a packet is covered with the checksum. It adds two socket options that you can change. self.setsockopt(IPPROTO\_UDPLITE, UDPLITE\_SEND\_CSCOV, length) will change what portion of outgoing packets are covered by the checksum and self.setsockopt(IPPROTO\_UDPLITE, UDPLITE\_RECV\_CSCOV, length) will filter out packets which cover too little of their data. In both cases length should be in range(8, 2\*\*16, 8).

Such a socket should be constructed with socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDPLITE) for IPv4 or socket(AF\_INET6, SOCK\_DGRAM, IPPROTO\_UDPLITE) for IPv6.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 2.6.20, FreeBSD >= 10.1

*Added in version 3.9.*

* [AF\_HYPERV](https://docs.python.org/3/library/socket.html#socket.AF_HYPERV) is a Windows-only socket based interface for communicating with Hyper-V hosts and guests. The address family is represented as a (vm\_id, service\_id) tuple where the vm\_id and service\_id are UUID strings.

The vm\_id is the virtual machine identifier or a set of known VMID values if the target is not a specific virtual machine. Known VMID constants defined on socket are:

* + HV\_GUID\_ZERO
  + HV\_GUID\_BROADCAST
  + HV\_GUID\_WILDCARD - Used to bind on itself and accept connections from all partitions.
  + HV\_GUID\_CHILDREN - Used to bind on itself and accept connection from child partitions.
  + HV\_GUID\_LOOPBACK - Used as a target to itself.
  + HV\_GUID\_PARENT - When used as a bind accepts connection from the parent partition. When used as an address target it will connect to the parent partition.

The service\_id is the service identifier of the registered service.

*Added in version 3.12.*

If you use a hostname in the *host* portion of IPv4/v6 socket address, the program may show a nondeterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in *host* portion.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised. Errors related to socket or address semantics raise [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError) or one of its subclasses.

Non-blocking mode is supported through [setblocking()](https://docs.python.org/3/library/socket.html" \l "socket.socket.setblocking" \o "socket.socket.setblocking). A generalization of this based on timeouts is supported through [settimeout()](https://docs.python.org/3/library/socket.html" \l "socket.socket.settimeout" \o "socket.socket.settimeout).

Module contents

The module [socket](https://docs.python.org/3/library/socket.html#module-socket) exports the following elements.

Exceptions

*exception*socket.**error**

A deprecated alias of [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError).

*Changed in version 3.3:*Following [**PEP 3151**](https://peps.python.org/pep-3151/), this class was made an alias of [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError).

*exception*socket.**herror**

A subclass of [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError), this exception is raised for address-related errors, i.e. for functions that use *h\_errno* in the POSIX C API, including [gethostbyname\_ex()](https://docs.python.org/3/library/socket.html" \l "socket.gethostbyname_ex" \o "socket.gethostbyname_ex) and [gethostbyaddr()](https://docs.python.org/3/library/socket.html" \l "socket.gethostbyaddr" \o "socket.gethostbyaddr). The accompanying value is a pair (h\_errno, string) representing an error returned by a library call. *h\_errno* is a numeric value, while *string* represents the description of *h\_errno*, as returned by the hstrerror() C function.

*Changed in version 3.3:*This class was made a subclass of [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError).

*exception*socket.**gaierror**

A subclass of [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError), this exception is raised for address-related errors by [getaddrinfo()](https://docs.python.org/3/library/socket.html" \l "socket.getaddrinfo" \o "socket.getaddrinfo) and [getnameinfo()](https://docs.python.org/3/library/socket.html" \l "socket.getnameinfo" \o "socket.getnameinfo). The accompanying value is a pair (error, string) representing an error returned by a library call. *string* represents the description of *error*, as returned by the gai\_strerror() C function. The numeric *error* value will match one of the EAI\_\* constants defined in this module.

*Changed in version 3.3:*This class was made a subclass of [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError).

*exception*socket.**timeout**

A deprecated alias of [TimeoutError](https://docs.python.org/3/library/exceptions.html" \l "TimeoutError" \o "TimeoutError).

A subclass of [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError), this exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to [settimeout()](https://docs.python.org/3/library/socket.html" \l "socket.socket.settimeout" \o "socket.socket.settimeout) (or implicitly through [setdefaulttimeout()](https://docs.python.org/3/library/socket.html" \l "socket.setdefaulttimeout" \o "socket.setdefaulttimeout)). The accompanying value is a string whose value is currently always “timed out”.

*Changed in version 3.3:*This class was made a subclass of [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError).

*Changed in version 3.10:*This class was made an alias of [TimeoutError](https://docs.python.org/3/library/exceptions.html" \l "TimeoutError" \o "TimeoutError).

Constants

The AF\_\* and SOCK\_\* constants are now AddressFamily and SocketKind [IntEnum](https://docs.python.org/3/library/enum.html" \l "enum.IntEnum" \o "enum.IntEnum) collections.

*Added in version 3.4.*

socket.**AF\_UNIX**

socket.**AF\_INET**

socket.**AF\_INET6**

These constants represent the address (and protocol) families, used for the first argument to [socket()](https://docs.python.org/3/library/socket.html#socket.socket). If the [AF\_UNIX](https://docs.python.org/3/library/socket.html#socket.AF_UNIX) constant is not defined then this protocol is unsupported. More constants may be available depending on the system.

socket.**AF\_UNSPEC**

[AF\_UNSPEC](https://docs.python.org/3/library/socket.html#socket.AF_UNSPEC) means that [getaddrinfo()](https://docs.python.org/3/library/socket.html" \l "socket.getaddrinfo" \o "socket.getaddrinfo) should return socket addresses for any address family (either IPv4, IPv6, or any other) that can be used.

socket.**SOCK\_STREAM**

socket.**SOCK\_DGRAM**

socket.**SOCK\_RAW**

socket.**SOCK\_RDM**

socket.**SOCK\_SEQPACKET**

These constants represent the socket types, used for the second argument to [socket()](https://docs.python.org/3/library/socket.html#socket.socket). More constants may be available depending on the system. (Only [SOCK\_STREAM](https://docs.python.org/3/library/socket.html#socket.SOCK_STREAM) and [SOCK\_DGRAM](https://docs.python.org/3/library/socket.html#socket.SOCK_DGRAM) appear to be generally useful.)

socket.**SOCK\_CLOEXEC**

socket.**SOCK\_NONBLOCK**

These two constants, if defined, can be combined with the socket types and allow you to set some flags atomically (thus avoiding possible race conditions and the need for separate calls).

**See also**

[Secure File Descriptor Handling](https://udrepper.livejournal.com/20407.html) for a more thorough explanation.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 2.6.27.

*Added in version 3.2.*

**SO\_\***

socket.**SOMAXCONN**

**MSG\_\***

**SOL\_\***

**SCM\_\***

**IPPROTO\_\***

**IPPORT\_\***

**INADDR\_\***

**IP\_\***

**IPV6\_\***

**EAI\_\***

**AI\_\***

**NI\_\***

**TCP\_\***

Many constants of these forms, documented in the Unix documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the [setsockopt()](https://docs.python.org/3/library/socket.html" \l "socket.socket.setsockopt" \o "socket.socket.setsockopt) and [getsockopt()](https://docs.python.org/3/library/socket.html" \l "socket.socket.getsockopt" \o "socket.socket.getsockopt) methods of socket objects. In most cases, only those symbols that are defined in the Unix header files are defined; for a few symbols, default values are provided.

*Changed in version 3.6:*SO\_DOMAIN, SO\_PROTOCOL, SO\_PEERSEC, SO\_PASSSEC, TCP\_USER\_TIMEOUT, TCP\_CONGESTION were added.

*Changed in version 3.6.5:*On Windows, TCP\_FASTOPEN, TCP\_KEEPCNT appear if run-time Windows supports.

*Changed in version 3.7:*TCP\_NOTSENT\_LOWAT was added.

On Windows, TCP\_KEEPIDLE, TCP\_KEEPINTVL appear if run-time Windows supports.

*Changed in version 3.10:*IP\_RECVTOS was added. Added TCP\_KEEPALIVE. On MacOS this constant can be used in the same way that TCP\_KEEPIDLE is used on Linux.

*Changed in version 3.11:*Added TCP\_CONNECTION\_INFO. On MacOS this constant can be used in the same way that TCP\_INFO is used on Linux and BSD.

*Changed in version 3.12:*Added SO\_RTABLE and SO\_USER\_COOKIE. On OpenBSD and FreeBSD respectively those constants can be used in the same way that SO\_MARK is used on Linux. Also added missing TCP socket options from Linux: TCP\_MD5SIG, TCP\_THIN\_LINEAR\_TIMEOUTS, TCP\_THIN\_DUPACK, TCP\_REPAIR, TCP\_REPAIR\_QUEUE, TCP\_QUEUE\_SEQ, TCP\_REPAIR\_OPTIONS, TCP\_TIMESTAMP, TCP\_CC\_INFO, TCP\_SAVE\_SYN, TCP\_SAVED\_SYN, TCP\_REPAIR\_WINDOW, TCP\_FASTOPEN\_CONNECT, TCP\_ULP, TCP\_MD5SIG\_EXT, TCP\_FASTOPEN\_KEY, TCP\_FASTOPEN\_NO\_COOKIE, TCP\_ZEROCOPY\_RECEIVE, TCP\_INQ, TCP\_TX\_DELAY. Added IP\_PKTINFO, IP\_UNBLOCK\_SOURCE, IP\_BLOCK\_SOURCE, IP\_ADD\_SOURCE\_MEMBERSHIP, IP\_DROP\_SOURCE\_MEMBERSHIP.

*Changed in version 3.13:*Added SO\_BINDTOIFINDEX. On Linux this constant can be used in the same way that SO\_BINDTODEVICE is used, but with the index of a network interface instead of its name.

socket.**AF\_CAN**

socket.**PF\_CAN**

**SOL\_CAN\_\***

**CAN\_\***

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 2.6.25, NetBSD >= 8.

*Added in version 3.3.*

*Changed in version 3.11:*NetBSD support was added.

*Changed in version 3.13.3 (unreleased):*Restored missing CAN\_RAW\_ERR\_FILTER on Linux.

socket.**CAN\_BCM**

**CAN\_BCM\_\***

CAN\_BCM, in the CAN protocol family, is the broadcast manager (BCM) protocol. Broadcast manager constants, documented in the Linux documentation, are also defined in the socket module.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 2.6.25.

**Note**

The CAN\_BCM\_CAN\_FD\_FRAME flag is only available on Linux >= 4.8.

*Added in version 3.4.*

socket.**CAN\_RAW\_FD\_FRAMES**

Enables CAN FD support in a CAN\_RAW socket. This is disabled by default. This allows your application to send both CAN and CAN FD frames; however, you must accept both CAN and CAN FD frames when reading from the socket.

This constant is documented in the Linux documentation.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 3.6.

*Added in version 3.5.*

socket.**CAN\_RAW\_JOIN\_FILTERS**

Joins the applied CAN filters such that only CAN frames that match all given CAN filters are passed to user space.

This constant is documented in the Linux documentation.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 4.1.

*Added in version 3.9.*

socket.**CAN\_ISOTP**

CAN\_ISOTP, in the CAN protocol family, is the ISO-TP (ISO 15765-2) protocol. ISO-TP constants, documented in the Linux documentation.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 2.6.25.

*Added in version 3.7.*

socket.**CAN\_J1939**

CAN\_J1939, in the CAN protocol family, is the SAE J1939 protocol. J1939 constants, documented in the Linux documentation.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 5.4.

*Added in version 3.9.*

socket.**AF\_DIVERT**

socket.**PF\_DIVERT**

These two constants, documented in the FreeBSD divert(4) manual page, are also defined in the socket module.

[Availability](https://docs.python.org/3/library/intro.html#availability): FreeBSD >= 14.0.

*Added in version 3.12.*

socket.**AF\_PACKET**

socket.**PF\_PACKET**

**PACKET\_\***

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 2.2.

socket.**ETH\_P\_ALL**

ETH\_P\_ALL can be used in the [socket](https://docs.python.org/3/library/socket.html#socket.socket) constructor as *proto* for the [AF\_PACKET](https://docs.python.org/3/library/socket.html#socket.AF_PACKET) family in order to capture every packet, regardless of protocol.

For more information, see the [*packet(7)*](https://manpages.debian.org/packet(7)) manpage.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux.

*Added in version 3.12.*

socket.**AF\_RDS**

socket.**PF\_RDS**

socket.**SOL\_RDS**

**RDS\_\***

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 2.6.30.

*Added in version 3.3.*

socket.**SIO\_RCVALL**

socket.**SIO\_KEEPALIVE\_VALS**

socket.**SIO\_LOOPBACK\_FAST\_PATH**

**RCVALL\_\***

Constants for Windows’ WSAIoctl(). The constants are used as arguments to the [ioctl()](https://docs.python.org/3/library/socket.html" \l "socket.socket.ioctl" \o "socket.socket.ioctl) method of socket objects.

*Changed in version 3.6:*SIO\_LOOPBACK\_FAST\_PATH was added.

**TIPC\_\***

TIPC related constants, matching the ones exported by the C socket API. See the TIPC documentation for more information.

socket.**AF\_ALG**

socket.**SOL\_ALG**

**ALG\_\***

Constants for Linux Kernel cryptography.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 2.6.38.

*Added in version 3.6.*

socket.**AF\_VSOCK**

socket.**IOCTL\_VM\_SOCKETS\_GET\_LOCAL\_CID**

**VMADDR\***

**SO\_VM\***

Constants for Linux host/guest communication.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 4.8.

*Added in version 3.7.*

socket.**AF\_LINK**

[Availability](https://docs.python.org/3/library/intro.html#availability): BSD, macOS.

*Added in version 3.4.*

socket.**has\_ipv6**

This constant contains a boolean value which indicates if IPv6 is supported on this platform.

socket.**BDADDR\_ANY**

socket.**BDADDR\_LOCAL**

These are string constants containing Bluetooth addresses with special meanings. For example, [BDADDR\_ANY](https://docs.python.org/3/library/socket.html#socket.BDADDR_ANY) can be used to indicate any address when specifying the binding socket with BTPROTO\_RFCOMM.

socket.**HCI\_FILTER**

socket.**HCI\_TIME\_STAMP**

socket.**HCI\_DATA\_DIR**

For use with BTPROTO\_HCI. HCI\_FILTER is only available on Linux and FreeBSD. HCI\_TIME\_STAMP and HCI\_DATA\_DIR are only available on Linux.

socket.**AF\_QIPCRTR**

Constant for Qualcomm’s IPC router protocol, used to communicate with service providing remote processors.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 4.7.

socket.**SCM\_CREDS2**

socket.**LOCAL\_CREDS**

socket.**LOCAL\_CREDS\_PERSISTENT**

LOCAL\_CREDS and LOCAL\_CREDS\_PERSISTENT can be used with SOCK\_DGRAM, SOCK\_STREAM sockets, equivalent to Linux/DragonFlyBSD SO\_PASSCRED, while LOCAL\_CREDS sends the credentials at first read, LOCAL\_CREDS\_PERSISTENT sends for each read, SCM\_CREDS2 must be then used for the latter for the message type.

*Added in version 3.11.*

[Availability](https://docs.python.org/3/library/intro.html#availability): FreeBSD.

socket.**SO\_INCOMING\_CPU**

Constant to optimize CPU locality, to be used in conjunction with SO\_REUSEPORT.

*Added in version 3.11.*

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 3.9

socket.**AF\_HYPERV**

socket.**HV\_PROTOCOL\_RAW**

socket.**HVSOCKET\_CONNECT\_TIMEOUT**

socket.**HVSOCKET\_CONNECT\_TIMEOUT\_MAX**

socket.**HVSOCKET\_CONNECTED\_SUSPEND**

socket.**HVSOCKET\_ADDRESS\_FLAG\_PASSTHRU**

socket.**HV\_GUID\_ZERO**

socket.**HV\_GUID\_WILDCARD**

socket.**HV\_GUID\_BROADCAST**

socket.**HV\_GUID\_CHILDREN**

socket.**HV\_GUID\_LOOPBACK**

socket.**HV\_GUID\_PARENT**

Constants for Windows Hyper-V sockets for host/guest communications.

[Availability](https://docs.python.org/3/library/intro.html#availability): Windows.

*Added in version 3.12.*

socket.**ETHERTYPE\_ARP**

socket.**ETHERTYPE\_IP**

socket.**ETHERTYPE\_IPV6**

socket.**ETHERTYPE\_VLAN**

[IEEE 802.3 protocol number](https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.txt). constants.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux, FreeBSD, macOS.

*Added in version 3.12.*

socket.**SHUT\_RD**

socket.**SHUT\_WR**

socket.**SHUT\_RDWR**

These constants are used by the [shutdown()](https://docs.python.org/3/library/socket.html#socket.socket.shutdown) method of socket objects.

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

Functions

Creating sockets

The following functions all create [socket objects](https://docs.python.org/3/library/socket.html#socket-objects).

*class*socket.**socket**(*family=AF\_INET*, *type=SOCK\_STREAM*, *proto=0*, *fileno=None*)

Create a new socket using the given address family, socket type and protocol number. The address family should be [AF\_INET](https://docs.python.org/3/library/socket.html#socket.AF_INET) (the default), [AF\_INET6](https://docs.python.org/3/library/socket.html#socket.AF_INET6), [AF\_UNIX](https://docs.python.org/3/library/socket.html#socket.AF_UNIX), [AF\_CAN](https://docs.python.org/3/library/socket.html#socket.AF_CAN), [AF\_PACKET](https://docs.python.org/3/library/socket.html#socket.AF_PACKET), or [AF\_RDS](https://docs.python.org/3/library/socket.html#socket.AF_RDS). The socket type should be [SOCK\_STREAM](https://docs.python.org/3/library/socket.html#socket.SOCK_STREAM) (the default), [SOCK\_DGRAM](https://docs.python.org/3/library/socket.html#socket.SOCK_DGRAM), [SOCK\_RAW](https://docs.python.org/3/library/socket.html#socket.SOCK_RAW) or perhaps one of the other SOCK\_ constants. The protocol number is usually zero and may be omitted or in the case where the address family is [AF\_CAN](https://docs.python.org/3/library/socket.html#socket.AF_CAN) the protocol should be one of CAN\_RAW, [CAN\_BCM](https://docs.python.org/3/library/socket.html#socket.CAN_BCM), [CAN\_ISOTP](https://docs.python.org/3/library/socket.html#socket.CAN_ISOTP) or [CAN\_J1939](https://docs.python.org/3/library/socket.html#socket.CAN_J1939).

If *fileno* is specified, the values for *family*, *type*, and *proto* are auto-detected from the specified file descriptor. Auto-detection can be overruled by calling the function with explicit *family*, *type*, or *proto* arguments. This only affects how Python represents e.g. the return value of [socket.getpeername()](https://docs.python.org/3/library/socket.html" \l "socket.socket.getpeername" \o "socket.socket.getpeername) but not the actual OS resource. Unlike [socket.fromfd()](https://docs.python.org/3/library/socket.html" \l "socket.fromfd" \o "socket.fromfd), *fileno* will return the same socket and not a duplicate. This may help close a detached socket using [socket.close()](https://docs.python.org/3/library/socket.html" \l "socket.close" \o "socket.close).

The newly created socket is [non-inheritable](https://docs.python.org/3/library/os.html#fd-inheritance).

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) socket.\_\_new\_\_ with arguments self, family, type, protocol.

*Changed in version 3.3:*The AF\_CAN family was added. The AF\_RDS family was added.

*Changed in version 3.4:*The CAN\_BCM protocol was added.

*Changed in version 3.4:*The returned socket is now non-inheritable.

*Changed in version 3.7:*The CAN\_ISOTP protocol was added.

*Changed in version 3.7:*When [SOCK\_NONBLOCK](https://docs.python.org/3/library/socket.html#socket.SOCK_NONBLOCK) or [SOCK\_CLOEXEC](https://docs.python.org/3/library/socket.html#socket.SOCK_CLOEXEC) bit flags are applied to *type* they are cleared, and [socket.type](https://docs.python.org/3/library/socket.html" \l "socket.socket.type" \o "socket.socket.type) will not reflect them. They are still passed to the underlying system socket() call. Therefore,

Copy

sock = socket.socket(

socket.AF\_INET,

socket.SOCK\_STREAM | socket.SOCK\_NONBLOCK)

will still create a non-blocking socket on OSes that support SOCK\_NONBLOCK, but sock.type will be set to socket.SOCK\_STREAM.

*Changed in version 3.9:*The CAN\_J1939 protocol was added.

*Changed in version 3.10:*The IPPROTO\_MPTCP protocol was added.

socket.**socketpair**([*family*[, *type*[, *proto*]]])

Build a pair of connected socket objects using the given address family, socket type, and protocol number. Address family, socket type, and protocol number are as for the [socket()](https://docs.python.org/3/library/socket.html#socket.socket) function above. The default family is [AF\_UNIX](https://docs.python.org/3/library/socket.html#socket.AF_UNIX) if defined on the platform; otherwise, the default is [AF\_INET](https://docs.python.org/3/library/socket.html#socket.AF_INET).

The newly created sockets are [non-inheritable](https://docs.python.org/3/library/os.html#fd-inheritance).

*Changed in version 3.2:*The returned socket objects now support the whole socket API, rather than a subset.

*Changed in version 3.4:*The returned sockets are now non-inheritable.

*Changed in version 3.5:*Windows support added.

socket.**create\_connection**(*address*, *timeout=GLOBAL\_DEFAULT*, *source\_address=None*, *\**, *all\_errors=False*)

Connect to a TCP service listening on the internet *address* (a 2-tuple (host, port)), and return the socket object. This is a higher-level function than [socket.connect()](https://docs.python.org/3/library/socket.html" \l "socket.socket.connect" \o "socket.socket.connect): if *host* is a non-numeric hostname, it will try to resolve it for both [AF\_INET](https://docs.python.org/3/library/socket.html#socket.AF_INET) and [AF\_INET6](https://docs.python.org/3/library/socket.html#socket.AF_INET6), and then try to connect to all possible addresses in turn until a connection succeeds. This makes it easy to write clients that are compatible to both IPv4 and IPv6.

Passing the optional *timeout* parameter will set the timeout on the socket instance before attempting to connect. If no *timeout* is supplied, the global default timeout setting returned by [getdefaulttimeout()](https://docs.python.org/3/library/socket.html" \l "socket.getdefaulttimeout" \o "socket.getdefaulttimeout) is used.

If supplied, *source\_address* must be a 2-tuple (host, port) for the socket to bind to as its source address before connecting. If host or port are ‘’ or 0 respectively the OS default behavior will be used.

When a connection cannot be created, an exception is raised. By default, it is the exception from the last address in the list. If *all\_errors* is True, it is an [ExceptionGroup](https://docs.python.org/3/library/exceptions.html" \l "ExceptionGroup" \o "ExceptionGroup) containing the errors of all attempts.

*Changed in version 3.2: source\_address* was added.

*Changed in version 3.11: all\_errors* was added.

socket.**create\_server**(*address*, *\**, *family=AF\_INET*, *backlog=None*, *reuse\_port=False*, *dualstack\_ipv6=False*)

Convenience function which creates a TCP socket bound to *address* (a 2-tuple (host, port)) and returns the socket object.

*family* should be either [AF\_INET](https://docs.python.org/3/library/socket.html#socket.AF_INET) or [AF\_INET6](https://docs.python.org/3/library/socket.html#socket.AF_INET6). *backlog* is the queue size passed to [socket.listen()](https://docs.python.org/3/library/socket.html" \l "socket.socket.listen" \o "socket.socket.listen); if not specified , a default reasonable value is chosen. *reuse\_port* dictates whether to set the SO\_REUSEPORT socket option.

If *dualstack\_ipv6* is true, *family* is [AF\_INET6](https://docs.python.org/3/library/socket.html#socket.AF_INET6) and the platform supports it the socket will be able to accept both IPv4 and IPv6 connections, else it will raise [ValueError](https://docs.python.org/3/library/exceptions.html" \l "ValueError" \o "ValueError). Most POSIX platforms and Windows are supposed to support this functionality. When this functionality is enabled the address returned by [socket.getpeername()](https://docs.python.org/3/library/socket.html" \l "socket.socket.getpeername" \o "socket.socket.getpeername) when an IPv4 connection occurs will be an IPv6 address represented as an IPv4-mapped IPv6 address. If *dualstack\_ipv6* is false it will explicitly disable this functionality on platforms that enable it by default (e.g. Linux). This parameter can be used in conjunction with [has\_dualstack\_ipv6()](https://docs.python.org/3/library/socket.html#socket.has_dualstack_ipv6):

Copy

**import** **socket**

addr = ("", 8080) *# all interfaces, port 8080*

**if** socket.has\_dualstack\_ipv6():

s = socket.create\_server(addr, family=socket.AF\_INET6, dualstack\_ipv6=**True**)

**else**:

s = socket.create\_server(addr)

**Note**

On POSIX platforms the SO\_REUSEADDR socket option is set in order to immediately reuse previous sockets which were bound on the same *address* and remained in TIME\_WAIT state.

*Added in version 3.8.*

socket.**has\_dualstack\_ipv6**()

Return True if the platform supports creating a TCP socket which can handle both IPv4 and IPv6 connections.

*Added in version 3.8.*

socket.**fromfd**(*fd*, *family*, *type*, *proto=0*)

Duplicate the file descriptor *fd* (an integer as returned by a file object’s [fileno()](https://docs.python.org/3/library/io.html" \l "io.IOBase.fileno" \o "io.IOBase.fileno) method) and build a socket object from the result. Address family, socket type and protocol number are as for the [socket()](https://docs.python.org/3/library/socket.html#socket.socket) function above. The file descriptor should refer to a socket, but this is not checked — subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (such as a server started by the Unix inet daemon). The socket is assumed to be in blocking mode.

The newly created socket is [non-inheritable](https://docs.python.org/3/library/os.html#fd-inheritance).

*Changed in version 3.4:*The returned socket is now non-inheritable.

socket.**fromshare**(*data*)

Instantiate a socket from data obtained from the [socket.share()](https://docs.python.org/3/library/socket.html" \l "socket.socket.share" \o "socket.socket.share) method. The socket is assumed to be in blocking mode.

[Availability](https://docs.python.org/3/library/intro.html#availability): Windows.

*Added in version 3.3.*

socket.**SocketType**

This is a Python type object that represents the socket object type. It is the same as type(socket(...)).

Other functions

The [socket](https://docs.python.org/3/library/socket.html#module-socket) module also offers various network-related services:

socket.**close**(*fd*)

Close a socket file descriptor. This is like [os.close()](https://docs.python.org/3/library/os.html" \l "os.close" \o "os.close), but for sockets. On some platforms (most noticeable Windows) [os.close()](https://docs.python.org/3/library/os.html" \l "os.close" \o "os.close) does not work for socket file descriptors.

*Added in version 3.7.*

socket.**getaddrinfo**(*host*, *port*, *family=AF\_UNSPEC*, *type=0*, *proto=0*, *flags=0*)

This function wraps the C function getaddrinfo of the underlying system.

Translate the *host*/*port* argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. *host* is a domain name, a string representation of an IPv4/v6 address or None. *port* is a string service name such as 'http', a numeric port number or None. By passing None as the value of *host* and *port*, you can pass NULL to the underlying C API.

The *family*, *type* and *proto* arguments can be optionally specified in order to provide options and limit the list of addresses returned. Pass their default values ([AF\_UNSPEC](https://docs.python.org/3/library/socket.html#socket.AF_UNSPEC), 0, and 0, respectively) to not limit the results. See the note below for details.

The *flags* argument can be one or several of the AI\_\* constants, and will influence how results are computed and returned. For example, AI\_NUMERICHOST will disable domain name resolution and will raise an error if *host* is a domain name.

The function returns a list of 5-tuples with the following structure:

(family, type, proto, canonname, sockaddr)

In these tuples, *family*, *type*, *proto* are all integers and are meant to be passed to the [socket()](https://docs.python.org/3/library/socket.html#socket.socket) function. *canonname* will be a string representing the canonical name of the *host* if AI\_CANONNAME is part of the *flags* argument; else *canonname* will be empty. *sockaddr* is a tuple describing a socket address, whose format depends on the returned *family* (a (address, port) 2-tuple for [AF\_INET](https://docs.python.org/3/library/socket.html#socket.AF_INET), a (address, port, flowinfo, scope\_id) 4-tuple for [AF\_INET6](https://docs.python.org/3/library/socket.html#socket.AF_INET6)), and is meant to be passed to the [socket.connect()](https://docs.python.org/3/library/socket.html" \l "socket.socket.connect" \o "socket.socket.connect) method.

**Note**

If you intend to use results from getaddrinfo() to create a socket (rather than, for example, retrieve *canonname*), consider limiting the results by *type* (e.g. [SOCK\_STREAM](https://docs.python.org/3/library/socket.html#socket.SOCK_STREAM) or [SOCK\_DGRAM](https://docs.python.org/3/library/socket.html#socket.SOCK_DGRAM)) and/or *proto* (e.g. IPPROTO\_TCP or IPPROTO\_UDP) that your application can handle.

The behavior with default values of *family*, *type*, *proto* and *flags* is system-specific.

Many systems (for example, most Linux configurations) will return a sorted list of all matching addresses. These addresses should generally be tried in order until a connection succeeds (possibly tried in parallel, for example, using a [Happy Eyeballs](https://en.wikipedia.org/wiki/Happy_Eyeballs) algorithm). In these cases, limiting the *type* and/or *proto* can help eliminate unsuccessful or unusable connection attempts.

Some systems will, however, only return a single address. (For example, this was reported on Solaris and AIX configurations.) On these systems, limiting the *type* and/or *proto* helps ensure that this address is usable.

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) socket.getaddrinfo with arguments host, port, family, type, protocol.

The following example fetches address information for a hypothetical TCP connection to example.org on port 80 (results may differ on your system if IPv6 isn’t enabled):

Copy

**>>>** socket.getaddrinfo("example.org", 80, proto=socket.IPPROTO\_TCP)

[(socket.AF\_INET6, socket.SOCK\_STREAM,

6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),

(socket.AF\_INET, socket.SOCK\_STREAM,

6, '', ('93.184.216.34', 80))]

*Changed in version 3.2:*parameters can now be passed using keyword arguments.

*Changed in version 3.7:*for IPv6 multicast addresses, string representing an address will not contain %scope\_id part.

socket.**getfqdn**([*name*])

Return a fully qualified domain name for *name*. If *name* is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by [gethostbyaddr()](https://docs.python.org/3/library/socket.html" \l "socket.gethostbyaddr" \o "socket.gethostbyaddr) is checked, followed by aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available and *name* was provided, it is returned unchanged. If *name* was empty or equal to '0.0.0.0', the hostname from [gethostname()](https://docs.python.org/3/library/socket.html" \l "socket.gethostname" \o "socket.gethostname) is returned.

socket.**gethostbyname**(*hostname*)

Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as '100.50.200.5'. If the host name is an IPv4 address itself it is returned unchanged. See [gethostbyname\_ex()](https://docs.python.org/3/library/socket.html" \l "socket.gethostbyname_ex" \o "socket.gethostbyname_ex) for a more complete interface. [gethostbyname()](https://docs.python.org/3/library/socket.html" \l "socket.gethostbyname" \o "socket.gethostbyname) does not support IPv6 name resolution, and [getaddrinfo()](https://docs.python.org/3/library/socket.html" \l "socket.getaddrinfo" \o "socket.getaddrinfo) should be used instead for IPv4/v6 dual stack support.

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) socket.gethostbyname with argument hostname.

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

socket.**gethostbyname\_ex**(*hostname*)

Translate a host name to IPv4 address format, extended interface. Return a 3-tuple (hostname, aliaslist, ipaddrlist) where *hostname* is the host’s primary host name, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4 addresses for the same interface on the same host (often but not always a single address). [gethostbyname\_ex()](https://docs.python.org/3/library/socket.html" \l "socket.gethostbyname_ex" \o "socket.gethostbyname_ex) does not support IPv6 name resolution, and [getaddrinfo()](https://docs.python.org/3/library/socket.html" \l "socket.getaddrinfo" \o "socket.getaddrinfo) should be used instead for IPv4/v6 dual stack support.

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) socket.gethostbyname with argument hostname.

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

socket.**gethostname**()

Return a string containing the hostname of the machine where the Python interpreter is currently executing.

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) socket.gethostname with no arguments.

Note: [gethostname()](https://docs.python.org/3/library/socket.html" \l "socket.gethostname" \o "socket.gethostname) doesn’t always return the fully qualified domain name; use [getfqdn()](https://docs.python.org/3/library/socket.html" \l "socket.getfqdn" \o "socket.getfqdn) for that.

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

socket.**gethostbyaddr**(*ip\_address*)

Return a 3-tuple (hostname, aliaslist, ipaddrlist) where *hostname* is the primary host name responding to the given *ip\_address*, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4/v6 addresses for the same interface on the same host (most likely containing only a single address). To find the fully qualified domain name, use the function [getfqdn()](https://docs.python.org/3/library/socket.html" \l "socket.getfqdn" \o "socket.getfqdn). [gethostbyaddr()](https://docs.python.org/3/library/socket.html" \l "socket.gethostbyaddr" \o "socket.gethostbyaddr) supports both IPv4 and IPv6.

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) socket.gethostbyaddr with argument ip\_address.

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

socket.**getnameinfo**(*sockaddr*, *flags*)

Translate a socket address *sockaddr* into a 2-tuple (host, port). Depending on the settings of *flags*, the result can contain a fully qualified domain name or numeric address representation in *host*. Similarly, *port* can contain a string port name or a numeric port number.

For IPv6 addresses, %scope\_id is appended to the host part if *sockaddr* contains meaningful *scope\_id*. Usually this happens for multicast addresses.

For more information about *flags* you can consult *[getnameinfo(3)](https://manpages.debian.org/getnameinfo(3))*.

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) socket.getnameinfo with argument sockaddr.

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

socket.**getprotobyname**(*protocolname*)

Translate an internet protocol name (for example, 'icmp') to a constant suitable for passing as the (optional) third argument to the [socket()](https://docs.python.org/3/library/socket.html#socket.socket) function. This is usually only needed for sockets opened in “raw” mode ([SOCK\_RAW](https://docs.python.org/3/library/socket.html#socket.SOCK_RAW)); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

socket.**getservbyname**(*servicename*[, *protocolname*])

Translate an internet service name and protocol name to a port number for that service. The optional protocol name, if given, should be 'tcp' or 'udp', otherwise any protocol will match.

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) socket.getservbyname with arguments servicename, protocolname.

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

socket.**getservbyport**(*port*[, *protocolname*])

Translate an internet port number and protocol name to a service name for that service. The optional protocol name, if given, should be 'tcp' or 'udp', otherwise any protocol will match.

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) socket.getservbyport with arguments port, protocolname.

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

socket.**ntohl**(*x*)

Convert 32-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.**ntohs**(*x*)

Convert 16-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

*Changed in version 3.10:*Raises [OverflowError](https://docs.python.org/3/library/exceptions.html" \l "OverflowError" \o "OverflowError) if *x* does not fit in a 16-bit unsigned integer.

socket.**htonl**(*x*)

Convert 32-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.**htons**(*x*)

Convert 16-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

*Changed in version 3.10:*Raises [OverflowError](https://docs.python.org/3/library/exceptions.html" \l "OverflowError" \o "OverflowError) if *x* does not fit in a 16-bit unsigned integer.

socket.**inet\_aton**(*ip\_string*)

Convert an IPv4 address from dotted-quad string format (for example, ‘123.45.67.89’) to 32-bit packed binary format, as a bytes object four characters in length. This is useful when conversing with a program that uses the standard C library and needs objects of type in\_addr, which is the C type for the 32-bit packed binary this function returns.

[inet\_aton()](https://docs.python.org/3/library/socket.html#socket.inet_aton) also accepts strings with less than three dots; see the Unix manual page *[inet(3)](https://manpages.debian.org/inet(3))* for details.

If the IPv4 address string passed to this function is invalid, [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError) will be raised. Note that exactly what is valid depends on the underlying C implementation of inet\_aton().

[inet\_aton()](https://docs.python.org/3/library/socket.html#socket.inet_aton) does not support IPv6, and [inet\_pton()](https://docs.python.org/3/library/socket.html" \l "socket.inet_pton" \o "socket.inet_pton) should be used instead for IPv4/v6 dual stack support.

socket.**inet\_ntoa**(*packed\_ip*)

Convert a 32-bit packed IPv4 address (a [bytes-like object](https://docs.python.org/3/glossary.html#term-bytes-like-object) four bytes in length) to its standard dotted-quad string representation (for example, ‘123.45.67.89’). This is useful when conversing with a program that uses the standard C library and needs objects of type in\_addr, which is the C type for the 32-bit packed binary data this function takes as an argument.

If the byte sequence passed to this function is not exactly 4 bytes in length, [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError) will be raised. [inet\_ntoa()](https://docs.python.org/3/library/socket.html" \l "socket.inet_ntoa" \o "socket.inet_ntoa) does not support IPv6, and [inet\_ntop()](https://docs.python.org/3/library/socket.html" \l "socket.inet_ntop" \o "socket.inet_ntop) should be used instead for IPv4/v6 dual stack support.

*Changed in version 3.5:*Writable [bytes-like object](https://docs.python.org/3/glossary.html#term-bytes-like-object) is now accepted.

socket.**inet\_pton**(*address\_family*, *ip\_string*)

Convert an IP address from its family-specific string format to a packed, binary format. [inet\_pton()](https://docs.python.org/3/library/socket.html" \l "socket.inet_pton" \o "socket.inet_pton) is useful when a library or network protocol calls for an object of type in\_addr (similar to [inet\_aton()](https://docs.python.org/3/library/socket.html" \l "socket.inet_aton" \o "socket.inet_aton)) or in6\_addr.

Supported values for *address\_family* are currently [AF\_INET](https://docs.python.org/3/library/socket.html#socket.AF_INET) and [AF\_INET6](https://docs.python.org/3/library/socket.html#socket.AF_INET6). If the IP address string *ip\_string* is invalid, [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError) will be raised. Note that exactly what is valid depends on both the value of *address\_family* and the underlying implementation of inet\_pton().

[Availability](https://docs.python.org/3/library/intro.html#availability): Unix, Windows.

*Changed in version 3.4:*Windows support added

socket.**inet\_ntop**(*address\_family*, *packed\_ip*)

Convert a packed IP address (a [bytes-like object](https://docs.python.org/3/glossary.html#term-bytes-like-object) of some number of bytes) to its standard, family-specific string representation (for example, '7.10.0.5' or '5aef:2b::8'). [inet\_ntop()](https://docs.python.org/3/library/socket.html" \l "socket.inet_ntop" \o "socket.inet_ntop) is useful when a library or network protocol returns an object of type in\_addr (similar to [inet\_ntoa()](https://docs.python.org/3/library/socket.html" \l "socket.inet_ntoa" \o "socket.inet_ntoa)) or in6\_addr.

Supported values for *address\_family* are currently [AF\_INET](https://docs.python.org/3/library/socket.html#socket.AF_INET) and [AF\_INET6](https://docs.python.org/3/library/socket.html#socket.AF_INET6). If the bytes object *packed\_ip* is not the correct length for the specified address family, [ValueError](https://docs.python.org/3/library/exceptions.html" \l "ValueError" \o "ValueError) will be raised. [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError) is raised for errors from the call to [inet\_ntop()](https://docs.python.org/3/library/socket.html" \l "socket.inet_ntop" \o "socket.inet_ntop).

[Availability](https://docs.python.org/3/library/intro.html#availability): Unix, Windows.

*Changed in version 3.4:*Windows support added

*Changed in version 3.5:*Writable [bytes-like object](https://docs.python.org/3/glossary.html#term-bytes-like-object) is now accepted.

socket.**CMSG\_LEN**(*length*)

Return the total length, without trailing padding, of an ancillary data item with associated data of the given *length*. This value can often be used as the buffer size for [recvmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recvmsg" \o "socket.socket.recvmsg) to receive a single item of ancillary data, but [**RFC 3542**](https://datatracker.ietf.org/doc/html/rfc3542.html) requires portable applications to use [CMSG\_SPACE()](https://docs.python.org/3/library/socket.html#socket.CMSG_SPACE) and thus include space for padding, even when the item will be the last in the buffer. Raises [OverflowError](https://docs.python.org/3/library/exceptions.html" \l "OverflowError" \o "OverflowError) if *length* is outside the permissible range of values.

[Availability](https://docs.python.org/3/library/intro.html#availability): Unix, not WASI.

Most Unix platforms.

*Added in version 3.3.*

socket.**CMSG\_SPACE**(*length*)

Return the buffer size needed for [recvmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recvmsg" \o "socket.socket.recvmsg) to receive an ancillary data item with associated data of the given *length*, along with any trailing padding. The buffer space needed to receive multiple items is the sum of the [CMSG\_SPACE()](https://docs.python.org/3/library/socket.html#socket.CMSG_SPACE) values for their associated data lengths. Raises [OverflowError](https://docs.python.org/3/library/exceptions.html" \l "OverflowError" \o "OverflowError) if *length* is outside the permissible range of values.

Note that some systems might support ancillary data without providing this function. Also note that setting the buffer size using the results of this function may not precisely limit the amount of ancillary data that can be received, since additional data may be able to fit into the padding area.

[Availability](https://docs.python.org/3/library/intro.html#availability): Unix, not WASI.

most Unix platforms.

*Added in version 3.3.*

socket.**getdefaulttimeout**()

Return the default timeout in seconds (float) for new socket objects. A value of None indicates that new socket objects have no timeout. When the socket module is first imported, the default is None.

socket.**setdefaulttimeout**(*timeout*)

Set the default timeout in seconds (float) for new socket objects. When the socket module is first imported, the default is None. See [settimeout()](https://docs.python.org/3/library/socket.html" \l "socket.socket.settimeout" \o "socket.socket.settimeout) for possible values and their respective meanings.

socket.**sethostname**(*name*)

Set the machine’s hostname to *name*. This will raise an [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError) if you don’t have enough rights.

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) socket.sethostname with argument name.

[Availability](https://docs.python.org/3/library/intro.html#availability): Unix, not Android.

*Added in version 3.3.*

socket.**if\_nameindex**()

Return a list of network interface information (index int, name string) tuples. [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError) if the system call fails.

[Availability](https://docs.python.org/3/library/intro.html#availability): Unix, Windows, not WASI.

*Added in version 3.3.*

*Changed in version 3.8:*Windows support was added.

**Note**

On Windows network interfaces have different names in different contexts (all names are examples):

* UUID: {FB605B73-AAC2-49A6-9A2F-25416AEA0573}
* name: ethernet\_32770
* friendly name: vEthernet (nat)
* description: Hyper-V Virtual Ethernet Adapter

This function returns names of the second form from the list, ethernet\_32770 in this example case.

socket.**if\_nametoindex**(*if\_name*)

Return a network interface index number corresponding to an interface name. [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError) if no interface with the given name exists.

[Availability](https://docs.python.org/3/library/intro.html#availability): Unix, Windows, not WASI.

*Added in version 3.3.*

*Changed in version 3.8:*Windows support was added.

**See also**

“Interface name” is a name as documented in [if\_nameindex()](https://docs.python.org/3/library/socket.html" \l "socket.if_nameindex" \o "socket.if_nameindex).

socket.**if\_indextoname**(*if\_index*)

Return a network interface name corresponding to an interface index number. [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError) if no interface with the given index exists.

[Availability](https://docs.python.org/3/library/intro.html#availability): Unix, Windows, not WASI.

*Added in version 3.3.*

*Changed in version 3.8:*Windows support was added.

**See also**

“Interface name” is a name as documented in [if\_nameindex()](https://docs.python.org/3/library/socket.html" \l "socket.if_nameindex" \o "socket.if_nameindex).

socket.**send\_fds**(*sock*, *buffers*, *fds*[, *flags*[, *address*]])

Send the list of file descriptors *fds* over an [AF\_UNIX](https://docs.python.org/3/library/socket.html#socket.AF_UNIX) socket *sock*. The *fds* parameter is a sequence of file descriptors. Consult [sendmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.sendmsg" \o "socket.socket.sendmsg) for the documentation of these parameters.

[Availability](https://docs.python.org/3/library/intro.html#availability): Unix, Windows, not WASI.

Unix platforms supporting [sendmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.sendmsg" \o "socket.socket.sendmsg) and SCM\_RIGHTS mechanism.

*Added in version 3.9.*

socket.**recv\_fds**(*sock*, *bufsize*, *maxfds*[, *flags*])

Receive up to *maxfds* file descriptors from an [AF\_UNIX](https://docs.python.org/3/library/socket.html#socket.AF_UNIX) socket *sock*. Return (msg, list(fds), flags, addr). Consult [recvmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recvmsg" \o "socket.socket.recvmsg) for the documentation of these parameters.

[Availability](https://docs.python.org/3/library/intro.html#availability): Unix, Windows, not WASI.

Unix platforms supporting [sendmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.sendmsg" \o "socket.socket.sendmsg) and SCM\_RIGHTS mechanism.

*Added in version 3.9.*

**Note**

Any truncated integers at the end of the list of file descriptors.

Socket Objects

Socket objects have the following methods. Except for [makefile()](https://docs.python.org/3/library/socket.html" \l "socket.socket.makefile" \o "socket.socket.makefile), these correspond to Unix system calls applicable to sockets.

*Changed in version 3.2:*Support for the [context manager](https://docs.python.org/3/glossary.html#term-context-manager) protocol was added. Exiting the context manager is equivalent to calling [close()](https://docs.python.org/3/library/socket.html#socket.close).

socket.**accept**()

Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair (conn, address) where *conn* is a *new* socket object usable to send and receive data on the connection, and *address* is the address bound to the socket on the other end of the connection.

The newly created socket is [non-inheritable](https://docs.python.org/3/library/os.html#fd-inheritance).

*Changed in version 3.4:*The socket is now non-inheritable.

*Changed in version 3.5:*If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an [InterruptedError](https://docs.python.org/3/library/exceptions.html" \l "InterruptedError" \o "InterruptedError) exception (see [**PEP 475**](https://peps.python.org/pep-0475/) for the rationale).

socket.**bind**(*address*)

Bind the socket to *address*. The socket must not already be bound. (The format of *address* depends on the address family — see above.)

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) socket.bind with arguments self, address.

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

socket.**close**()

Mark the socket closed. The underlying system resource (e.g. a file descriptor) is also closed when all file objects from [makefile()](https://docs.python.org/3/library/socket.html" \l "socket.socket.makefile" \o "socket.socket.makefile) are closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed).

Sockets are automatically closed when they are garbage-collected, but it is recommended to [close()](https://docs.python.org/3/library/socket.html#socket.close) them explicitly, or to use a [with](https://docs.python.org/3/reference/compound_stmts.html#with) statement around them.

*Changed in version 3.6:*[OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError) is now raised if an error occurs when the underlying close() call is made.

**Note**

[close()](https://docs.python.org/3/library/socket.html#socket.close) releases the resource associated with a connection but does not necessarily close the connection immediately. If you want to close the connection in a timely fashion, call [shutdown()](https://docs.python.org/3/library/socket.html#socket.socket.shutdown) before [close()](https://docs.python.org/3/library/socket.html#socket.close).

socket.**connect**(*address*)

Connect to a remote socket at *address*. (The format of *address* depends on the address family — see above.)

If the connection is interrupted by a signal, the method waits until the connection completes, or raise a [TimeoutError](https://docs.python.org/3/library/exceptions.html" \l "TimeoutError" \o "TimeoutError) on timeout, if the signal handler doesn’t raise an exception and the socket is blocking or has a timeout. For non-blocking sockets, the method raises an [InterruptedError](https://docs.python.org/3/library/exceptions.html" \l "InterruptedError" \o "InterruptedError) exception if the connection is interrupted by a signal (or the exception raised by the signal handler).

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) socket.connect with arguments self, address.

*Changed in version 3.5:*The method now waits until the connection completes instead of raising an [InterruptedError](https://docs.python.org/3/library/exceptions.html" \l "InterruptedError" \o "InterruptedError) exception if the connection is interrupted by a signal, the signal handler doesn’t raise an exception and the socket is blocking or has a timeout (see the [**PEP 475**](https://peps.python.org/pep-0475/) for the rationale).

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

socket.**connect\_ex**(*address*)

Like connect(address), but return an error indicator instead of raising an exception for errors returned by the C-level connect() call (other problems, such as “host not found,” can still raise exceptions). The error indicator is 0 if the operation succeeded, otherwise the value of the errno variable. This is useful to support, for example, asynchronous connects.

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) socket.connect with arguments self, address.

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

socket.**detach**()

Put the socket object into closed state without actually closing the underlying file descriptor. The file descriptor is returned, and can be reused for other purposes.

*Added in version 3.2.*

socket.**dup**()

Duplicate the socket.

The newly created socket is [non-inheritable](https://docs.python.org/3/library/os.html#fd-inheritance).

*Changed in version 3.4:*The socket is now non-inheritable.

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

socket.**fileno**()

Return the socket’s file descriptor (a small integer), or -1 on failure. This is useful with [select.select()](https://docs.python.org/3/library/select.html" \l "select.select" \o "select.select).

Under Windows the small integer returned by this method cannot be used where a file descriptor can be used (such as [os.fdopen()](https://docs.python.org/3/library/os.html" \l "os.fdopen" \o "os.fdopen)). Unix does not have this limitation.

socket.**get\_inheritable**()

Get the [inheritable flag](https://docs.python.org/3/library/os.html#fd-inheritance) of the socket’s file descriptor or socket’s handle: True if the socket can be inherited in child processes, False if it cannot.

*Added in version 3.4.*

socket.**getpeername**()

Return the remote address to which the socket is connected. This is useful to find out the port number of a remote IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.) On some systems this function is not supported.

socket.**getsockname**()

Return the socket’s own address. This is useful to find out the port number of an IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.)

socket.**getsockopt**(*level*, *optname*[, *buflen*])

Return the value of the given socket option (see the Unix man page *[getsockopt(2)](https://manpages.debian.org/getsockopt(2))*). The needed symbolic constants ([SO\_\* etc.](https://docs.python.org/3/library/socket.html#socket-unix-constants)) are defined in this module. If *buflen* is absent, an integer option is assumed and its integer value is returned by the function. If *buflen* is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a bytes object. It is up to the caller to decode the contents of the buffer (see the optional built-in module [struct](https://docs.python.org/3/library/struct.html#module-struct) for a way to decode C structures encoded as byte strings).

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

socket.**getblocking**()

Return True if socket is in blocking mode, False if in non-blocking.

This is equivalent to checking socket.gettimeout() != 0.

*Added in version 3.7.*

socket.**gettimeout**()

Return the timeout in seconds (float) associated with socket operations, or None if no timeout is set. This reflects the last call to [setblocking()](https://docs.python.org/3/library/socket.html" \l "socket.socket.setblocking" \o "socket.socket.setblocking) or [settimeout()](https://docs.python.org/3/library/socket.html" \l "socket.socket.settimeout" \o "socket.socket.settimeout).

socket.**ioctl**(*control*, *option*)

**Platform:**

Windows

The [ioctl()](https://docs.python.org/3/library/socket.html" \l "socket.socket.ioctl" \o "socket.socket.ioctl) method is a limited interface to the WSAIoctl system interface. Please refer to the [Win32 documentation](https://msdn.microsoft.com/en-us/library/ms741621%28VS.85%29.aspx) for more information.

On other platforms, the generic [fcntl.fcntl()](https://docs.python.org/3/library/fcntl.html" \l "fcntl.fcntl" \o "fcntl.fcntl) and [fcntl.ioctl()](https://docs.python.org/3/library/fcntl.html" \l "fcntl.ioctl" \o "fcntl.ioctl) functions may be used; they accept a socket object as their first argument.

Currently only the following control codes are supported: SIO\_RCVALL, SIO\_KEEPALIVE\_VALS, and SIO\_LOOPBACK\_FAST\_PATH.

*Changed in version 3.6:*SIO\_LOOPBACK\_FAST\_PATH was added.

socket.**listen**([*backlog*])

Enable a server to accept connections. If *backlog* is specified, it must be at least 0 (if it is lower, it is set to 0); it specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

*Changed in version 3.5:*The *backlog* parameter is now optional.

socket.**makefile**(*mode='r'*, *buffering=None*, *\**, *encoding=None*, *errors=None*, *newline=None*)

Return a [file object](https://docs.python.org/3/glossary.html#term-file-object) associated with the socket. The exact returned type depends on the arguments given to [makefile()](https://docs.python.org/3/library/socket.html" \l "socket.socket.makefile" \o "socket.socket.makefile). These arguments are interpreted the same way as by the built-in [open()](https://docs.python.org/3/library/functions.html#open) function, except the only supported *mode* values are 'r' (default), 'w', 'b', or a combination of those.

The socket must be in blocking mode; it can have a timeout, but the file object’s internal buffer may end up in an inconsistent state if a timeout occurs.

Closing the file object returned by [makefile()](https://docs.python.org/3/library/socket.html" \l "socket.socket.makefile" \o "socket.socket.makefile) won’t close the original socket unless all other file objects have been closed and [socket.close()](https://docs.python.org/3/library/socket.html" \l "socket.close" \o "socket.close) has been called on the socket object.

**Note**

On Windows, the file-like object created by [makefile()](https://docs.python.org/3/library/socket.html" \l "socket.socket.makefile" \o "socket.socket.makefile) cannot be used where a file object with a file descriptor is expected, such as the stream arguments of [subprocess.Popen()](https://docs.python.org/3/library/subprocess.html" \l "subprocess.Popen" \o "subprocess.Popen).

socket.**recv**(*bufsize*[, *flags*])

Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by *bufsize*. A returned empty bytes object indicates that the client has disconnected. See the Unix manual page *[recv(2)](https://manpages.debian.org/recv(2))* for the meaning of the optional argument *flags*; it defaults to zero.

*Changed in version 3.5:*If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an [InterruptedError](https://docs.python.org/3/library/exceptions.html" \l "InterruptedError" \o "InterruptedError) exception (see [**PEP 475**](https://peps.python.org/pep-0475/) for the rationale).

socket.**recvfrom**(*bufsize*[, *flags*])

Receive data from the socket. The return value is a pair (bytes, address) where *bytes* is a bytes object representing the data received and *address* is the address of the socket sending the data. See the Unix manual page *[recv(2)](https://manpages.debian.org/recv(2))* for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family — see above.)

*Changed in version 3.5:*If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an [InterruptedError](https://docs.python.org/3/library/exceptions.html" \l "InterruptedError" \o "InterruptedError) exception (see [**PEP 475**](https://peps.python.org/pep-0475/) for the rationale).

*Changed in version 3.7:*For multicast IPv6 address, first item of *address* does not contain %scope\_id part anymore. In order to get full IPv6 address use [getnameinfo()](https://docs.python.org/3/library/socket.html" \l "socket.getnameinfo" \o "socket.getnameinfo).

socket.**recvmsg**(*bufsize*[, *ancbufsize*[, *flags*]])

Receive normal data (up to *bufsize* bytes) and ancillary data from the socket. The *ancbufsize* argument sets the size in bytes of the internal buffer used to receive the ancillary data; it defaults to 0, meaning that no ancillary data will be received. Appropriate buffer sizes for ancillary data can be calculated using [CMSG\_SPACE()](https://docs.python.org/3/library/socket.html#socket.CMSG_SPACE) or [CMSG\_LEN()](https://docs.python.org/3/library/socket.html#socket.CMSG_LEN), and items which do not fit into the buffer might be truncated or discarded. The *flags* argument defaults to 0 and has the same meaning as for [recv()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recv" \o "socket.socket.recv).

The return value is a 4-tuple: (data, ancdata, msg\_flags, address). The *data* item is a [bytes](https://docs.python.org/3/library/stdtypes.html#bytes) object holding the non-ancillary data received. The *ancdata* item is a list of zero or more tuples (cmsg\_level, cmsg\_type, cmsg\_data) representing the ancillary data (control messages) received: *cmsg\_level* and *cmsg\_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg\_data* is a [bytes](https://docs.python.org/3/library/stdtypes.html#bytes) object holding the associated data. The *msg\_flags* item is the bitwise OR of various flags indicating conditions on the received message; see your system documentation for details. If the receiving socket is unconnected, *address* is the address of the sending socket, if available; otherwise, its value is unspecified.

On some systems, [sendmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.sendmsg" \o "socket.socket.sendmsg) and [recvmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recvmsg" \o "socket.socket.recvmsg) can be used to pass file descriptors between processes over an [AF\_UNIX](https://docs.python.org/3/library/socket.html#socket.AF_UNIX) socket. When this facility is used (it is often restricted to [SOCK\_STREAM](https://docs.python.org/3/library/socket.html#socket.SOCK_STREAM) sockets), [recvmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recvmsg" \o "socket.socket.recvmsg) will return, in its ancillary data, items of the form (socket.SOL\_SOCKET, socket.SCM\_RIGHTS, fds), where *fds* is a [bytes](https://docs.python.org/3/library/stdtypes.html#bytes) object representing the new file descriptors as a binary array of the native C int type. If [recvmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recvmsg" \o "socket.socket.recvmsg) raises an exception after the system call returns, it will first attempt to close any file descriptors received via this mechanism.

Some systems do not indicate the truncated length of ancillary data items which have been only partially received. If an item appears to extend beyond the end of the buffer, [recvmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recvmsg" \o "socket.socket.recvmsg) will issue a [RuntimeWarning](https://docs.python.org/3/library/exceptions.html#RuntimeWarning), and will return the part of it which is inside the buffer provided it has not been truncated before the start of its associated data.

On systems which support the SCM\_RIGHTS mechanism, the following function will receive up to *maxfds* file descriptors, returning the message data and a list containing the descriptors (while ignoring unexpected conditions such as unrelated control messages being received). See also [sendmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.sendmsg" \o "socket.socket.sendmsg).

Copy

**import** **socket**, **array**

**def** recv\_fds(sock, msglen, maxfds):

fds = array.array("i") *# Array of ints*

msg, ancdata, flags, addr = sock.recvmsg(msglen, socket.CMSG\_LEN(maxfds \* fds.itemsize))

**for** cmsg\_level, cmsg\_type, cmsg\_data **in** ancdata:

**if** cmsg\_level == socket.SOL\_SOCKET **and** cmsg\_type == socket.SCM\_RIGHTS:

*# Append data, ignoring any truncated integers at the end.*

fds.frombytes(cmsg\_data[:len(cmsg\_data) - (len(cmsg\_data) % fds.itemsize)])

**return** msg, list(fds)

[Availability](https://docs.python.org/3/library/intro.html#availability): Unix.

Most Unix platforms.

*Added in version 3.3.*

*Changed in version 3.5:*If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an [InterruptedError](https://docs.python.org/3/library/exceptions.html" \l "InterruptedError" \o "InterruptedError) exception (see [**PEP 475**](https://peps.python.org/pep-0475/) for the rationale).

socket.**recvmsg\_into**(*buffers*[, *ancbufsize*[, *flags*]])

Receive normal data and ancillary data from the socket, behaving as [recvmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recvmsg" \o "socket.socket.recvmsg) would, but scatter the non-ancillary data into a series of buffers instead of returning a new bytes object. The *buffers* argument must be an iterable of objects that export writable buffers (e.g. [bytearray](https://docs.python.org/3/library/stdtypes.html" \l "bytearray" \o "bytearray) objects); these will be filled with successive chunks of the non-ancillary data until it has all been written or there are no more buffers. The operating system may set a limit ([sysconf()](https://docs.python.org/3/library/os.html" \l "os.sysconf" \o "os.sysconf) value SC\_IOV\_MAX) on the number of buffers that can be used. The *ancbufsize* and *flags* arguments have the same meaning as for [recvmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recvmsg" \o "socket.socket.recvmsg).

The return value is a 4-tuple: (nbytes, ancdata, msg\_flags, address), where *nbytes* is the total number of bytes of non-ancillary data written into the buffers, and *ancdata*, *msg\_flags* and *address* are the same as for [recvmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recvmsg" \o "socket.socket.recvmsg).

Example:

Copy

**>>> import** **socket**

**>>>** s1, s2 = socket.socketpair()

**>>>** b1 = bytearray(b'----')

**>>>** b2 = bytearray(b'0123456789')

**>>>** b3 = bytearray(b'--------------')

**>>>** s1.send(b'Mary had a little lamb')

22

**>>>** s2.recvmsg\_into([b1, memoryview(b2)[2:9], b3])

(22, [], 0, None)

**>>>** [b1, b2, b3]

[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'little lamb---')]

[Availability](https://docs.python.org/3/library/intro.html#availability): Unix.

Most Unix platforms.

*Added in version 3.3.*

socket.**recvfrom\_into**(*buffer*[, *nbytes*[, *flags*]])

Receive data from the socket, writing it into *buffer* instead of creating a new bytestring. The return value is a pair (nbytes, address) where *nbytes* is the number of bytes received and *address* is the address of the socket sending the data. See the Unix manual page *[recv(2)](https://manpages.debian.org/recv(2))* for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family — see above.)

socket.**recv\_into**(*buffer*[, *nbytes*[, *flags*]])

Receive up to *nbytes* bytes from the socket, storing the data into a buffer rather than creating a new bytestring. If *nbytes* is not specified (or 0), receive up to the size available in the given buffer. Returns the number of bytes received. See the Unix manual page *[recv(2)](https://manpages.debian.org/recv(2))* for the meaning of the optional argument *flags*; it defaults to zero.

socket.**send**(*bytes*[, *flags*])

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for [recv()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recv" \o "socket.socket.recv) above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data. For further information on this topic, consult the [Socket Programming HOWTO](https://docs.python.org/3/howto/sockets.html#socket-howto).

*Changed in version 3.5:*If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an [InterruptedError](https://docs.python.org/3/library/exceptions.html" \l "InterruptedError" \o "InterruptedError) exception (see [**PEP 475**](https://peps.python.org/pep-0475/) for the rationale).

socket.**sendall**(*bytes*[, *flags*])

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for [recv()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recv" \o "socket.socket.recv) above. Unlike [send()](https://docs.python.org/3/library/socket.html#socket.socket.send), this method continues to send data from *bytes* until either all data has been sent or an error occurs. None is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

*Changed in version 3.5:*The socket timeout is no longer reset each time data is sent successfully. The socket timeout is now the maximum total duration to send all data.

*Changed in version 3.5:*If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an [InterruptedError](https://docs.python.org/3/library/exceptions.html" \l "InterruptedError" \o "InterruptedError) exception (see [**PEP 475**](https://peps.python.org/pep-0475/) for the rationale).

socket.**sendto**(*bytes*, *address*)

socket.**sendto**(*bytes*, *flags*, *address*)

Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*. The optional *flags* argument has the same meaning as for [recv()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recv" \o "socket.socket.recv) above. Return the number of bytes sent. (The format of *address* depends on the address family — see above.)

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) socket.sendto with arguments self, address.

*Changed in version 3.5:*If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an [InterruptedError](https://docs.python.org/3/library/exceptions.html" \l "InterruptedError" \o "InterruptedError) exception (see [**PEP 475**](https://peps.python.org/pep-0475/) for the rationale).

socket.**sendmsg**(*buffers*[, *ancdata*[, *flags*[, *address*]]])

Send normal and ancillary data to the socket, gathering the non-ancillary data from a series of buffers and concatenating it into a single message. The *buffers* argument specifies the non-ancillary data as an iterable of [bytes-like objects](https://docs.python.org/3/glossary.html#term-bytes-like-object) (e.g. [bytes](https://docs.python.org/3/library/stdtypes.html#bytes) objects); the operating system may set a limit ([sysconf()](https://docs.python.org/3/library/os.html" \l "os.sysconf" \o "os.sysconf) value SC\_IOV\_MAX) on the number of buffers that can be used. The *ancdata* argument specifies the ancillary data (control messages) as an iterable of zero or more tuples (cmsg\_level, cmsg\_type, cmsg\_data), where *cmsg\_level* and *cmsg\_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg\_data* is a bytes-like object holding the associated data. Note that some systems (in particular, systems without [CMSG\_SPACE()](https://docs.python.org/3/library/socket.html#socket.CMSG_SPACE)) might support sending only one control message per call. The *flags* argument defaults to 0 and has the same meaning as for [send()](https://docs.python.org/3/library/socket.html#socket.socket.send). If *address* is supplied and not None, it sets a destination address for the message. The return value is the number of bytes of non-ancillary data sent.

The following function sends the list of file descriptors *fds* over an [AF\_UNIX](https://docs.python.org/3/library/socket.html#socket.AF_UNIX) socket, on systems which support the SCM\_RIGHTS mechanism. See also [recvmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recvmsg" \o "socket.socket.recvmsg).

Copy

**import** **socket**, **array**

**def** send\_fds(sock, msg, fds):

**return** sock.sendmsg([msg], [(socket.SOL\_SOCKET, socket.SCM\_RIGHTS, array.array("i", fds))])

[Availability](https://docs.python.org/3/library/intro.html#availability): Unix, not WASI.

Most Unix platforms.

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) socket.sendmsg with arguments self, address.

*Added in version 3.3.*

*Changed in version 3.5:*If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an [InterruptedError](https://docs.python.org/3/library/exceptions.html" \l "InterruptedError" \o "InterruptedError) exception (see [**PEP 475**](https://peps.python.org/pep-0475/) for the rationale).

socket.**sendmsg\_afalg**([*msg*, ]*\**, *op*[, *iv*[, *assoclen*[, *flags*]]])

Specialized version of [sendmsg()](https://docs.python.org/3/library/socket.html" \l "socket.socket.sendmsg" \o "socket.socket.sendmsg) for [AF\_ALG](https://docs.python.org/3/library/socket.html#socket.AF_ALG) socket. Set mode, IV, AEAD associated data length and flags for [AF\_ALG](https://docs.python.org/3/library/socket.html#socket.AF_ALG) socket.

[Availability](https://docs.python.org/3/library/intro.html#availability): Linux >= 2.6.38.

*Added in version 3.6.*

socket.**sendfile**(*file*, *offset=0*, *count=None*)

Send a file until EOF is reached by using high-performance [os.sendfile](https://docs.python.org/3/library/os.html" \l "os.sendfile" \o "os.sendfile) and return the total number of bytes which were sent. *file* must be a regular file object opened in binary mode. If [os.sendfile](https://docs.python.org/3/library/os.html" \l "os.sendfile" \o "os.sendfile) is not available (e.g. Windows) or *file* is not a regular file [send()](https://docs.python.org/3/library/socket.html#socket.socket.send) will be used instead. *offset* tells from where to start reading the file. If specified, *count* is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is updated on return or also in case of error in which case [file.tell()](https://docs.python.org/3/library/io.html" \l "io.IOBase.tell" \o "io.IOBase.tell) can be used to figure out the number of bytes which were sent. The socket must be of [SOCK\_STREAM](https://docs.python.org/3/library/socket.html#socket.SOCK_STREAM) type. Non-blocking sockets are not supported.

*Added in version 3.5.*

socket.**set\_inheritable**(*inheritable*)

Set the [inheritable flag](https://docs.python.org/3/library/os.html#fd-inheritance) of the socket’s file descriptor or socket’s handle.

*Added in version 3.4.*

socket.**setblocking**(*flag*)

Set blocking or non-blocking mode of the socket: if *flag* is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain [settimeout()](https://docs.python.org/3/library/socket.html" \l "socket.socket.settimeout" \o "socket.socket.settimeout) calls:

* sock.setblocking(True) is equivalent to sock.settimeout(None)
* sock.setblocking(False) is equivalent to sock.settimeout(0.0)

*Changed in version 3.7:*The method no longer applies [SOCK\_NONBLOCK](https://docs.python.org/3/library/socket.html#socket.SOCK_NONBLOCK) flag on [socket.type](https://docs.python.org/3/library/socket.html" \l "socket.socket.type" \o "socket.socket.type).

socket.**settimeout**(*value*)

Set a timeout on blocking socket operations. The *value* argument can be a nonnegative floating-point number expressing seconds, or None. If a non-zero value is given, subsequent socket operations will raise a [timeout](https://docs.python.org/3/library/socket.html#socket.timeout) exception if the timeout period *value* has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If None is given, the socket is put in blocking mode.

For further information, please consult the [notes on socket timeouts](https://docs.python.org/3/library/socket.html#socket-timeouts).

*Changed in version 3.7:*The method no longer toggles [SOCK\_NONBLOCK](https://docs.python.org/3/library/socket.html#socket.SOCK_NONBLOCK) flag on [socket.type](https://docs.python.org/3/library/socket.html" \l "socket.socket.type" \o "socket.socket.type).

socket.**setsockopt**(*level*, *optname*, *value:*[*int*](https://docs.python.org/3/library/functions.html#int))

socket.**setsockopt**(*level*, *optname*, *value: buffer*)

socket.**setsockopt**(*level*, *optname*, *None*, *optlen: int*)

Set the value of the given socket option (see the Unix manual page *[setsockopt(2)](https://manpages.debian.org/setsockopt(2))*). The needed symbolic constants are defined in this module (SO\_\* etc. <socket-unix-constants>). The value can be an integer, None or a [bytes-like object](https://docs.python.org/3/glossary.html#term-bytes-like-object) representing a buffer. In the later case it is up to the caller to ensure that the bytestring contains the proper bits (see the optional built-in module [struct](https://docs.python.org/3/library/struct.html#module-struct) for a way to encode C structures as bytestrings). When *value* is set to None, *optlen* argument is required. It’s equivalent to call setsockopt() C function with optval=NULL and optlen=optlen.

*Changed in version 3.5:*Writable [bytes-like object](https://docs.python.org/3/glossary.html#term-bytes-like-object) is now accepted.

*Changed in version 3.6:*setsockopt(level, optname, None, optlen: int) form added.

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

socket.**shutdown**(*how*)

Shut down one or both halves of the connection. If *how* is [SHUT\_RD](https://docs.python.org/3/library/socket.html#socket.SHUT_RD), further receives are disallowed. If *how* is [SHUT\_WR](https://docs.python.org/3/library/socket.html#socket.SHUT_WR), further sends are disallowed. If *how* is [SHUT\_RDWR](https://docs.python.org/3/library/socket.html#socket.SHUT_RDWR), further sends and receives are disallowed.

[Availability](https://docs.python.org/3/library/intro.html#availability): not WASI.

socket.**share**(*process\_id*)

Duplicate a socket and prepare it for sharing with a target process. The target process must be provided with *process\_id*. The resulting bytes object can then be passed to the target process using some form of interprocess communication and the socket can be recreated there using [fromshare()](https://docs.python.org/3/library/socket.html" \l "socket.fromshare" \o "socket.fromshare). Once this method has been called, it is safe to close the socket since the operating system has already duplicated it for the target process.

[Availability](https://docs.python.org/3/library/intro.html#availability): Windows.

*Added in version 3.3.*

Note that there are no methods read() or write(); use [recv()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recv" \o "socket.socket.recv) and [send()](https://docs.python.org/3/library/socket.html#socket.socket.send) without *flags* argument instead.

Socket objects also have these (read-only) attributes that correspond to the values given to the [socket](https://docs.python.org/3/library/socket.html#socket.socket) constructor.

socket.**family**

The socket family.

socket.**type**

The socket type.

socket.**proto**

The socket protocol.

Notes on socket timeouts

A socket object can be in one of three modes: blocking, non-blocking, or timeout. Sockets are by default always created in blocking mode, but this can be changed by calling [setdefaulttimeout()](https://docs.python.org/3/library/socket.html" \l "socket.setdefaulttimeout" \o "socket.setdefaulttimeout).

* In *blocking mode*, operations block until complete or the system returns an error (such as connection timed out).
* In *non-blocking mode*, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately: functions from the [select](https://docs.python.org/3/library/select.html#module-select) module can be used to know when and whether a socket is available for reading or writing.
* In *timeout mode*, operations fail if they cannot be completed within the timeout specified for the socket (they raise a [timeout](https://docs.python.org/3/library/socket.html#socket.timeout) exception) or if the system returns an error.

**Note**

At the operating system level, sockets in *timeout mode* are internally set in non-blocking mode. Also, the blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. This implementation detail can have visible consequences if e.g. you decide to use the [fileno()](https://docs.python.org/3/library/socket.html" \l "socket.socket.fileno" \o "socket.socket.fileno) of a socket.

Timeouts and the connect method

The [connect()](https://docs.python.org/3/library/socket.html#socket.socket.connect) operation is also subject to the timeout setting, and in general it is recommended to call [settimeout()](https://docs.python.org/3/library/socket.html" \l "socket.socket.settimeout" \o "socket.socket.settimeout) before calling [connect()](https://docs.python.org/3/library/socket.html#socket.socket.connect) or pass a timeout parameter to [create\_connection()](https://docs.python.org/3/library/socket.html" \l "socket.create_connection" \o "socket.create_connection). However, the system network stack may also return a connection timeout error of its own regardless of any Python socket timeout setting.

Timeouts and the accept method

If [getdefaulttimeout()](https://docs.python.org/3/library/socket.html" \l "socket.getdefaulttimeout" \o "socket.getdefaulttimeout) is not [None](https://docs.python.org/3/library/constants.html#None), sockets returned by the [accept()](https://docs.python.org/3/library/socket.html#socket.socket.accept) method inherit that timeout. Otherwise, the behaviour depends on settings of the listening socket:

* if the listening socket is in *blocking mode* or in *timeout mode*, the socket returned by [accept()](https://docs.python.org/3/library/socket.html#socket.socket.accept) is in *blocking mode*;
* if the listening socket is in *non-blocking mode*, whether the socket returned by [accept()](https://docs.python.org/3/library/socket.html#socket.socket.accept) is in blocking or non-blocking mode is operating system-dependent. If you want to ensure cross-platform behaviour, it is recommended you manually override this setting.

Example

Here are four minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence [socket()](https://docs.python.org/3/library/socket.html#socket.socket), [bind()](https://docs.python.org/3/library/socket.html#socket.socket.bind), [listen()](https://docs.python.org/3/library/socket.html#socket.socket.listen), [accept()](https://docs.python.org/3/library/socket.html#socket.socket.accept) (possibly repeating the [accept()](https://docs.python.org/3/library/socket.html#socket.socket.accept) to service more than one client), while a client only needs the sequence [socket()](https://docs.python.org/3/library/socket.html#socket.socket), [connect()](https://docs.python.org/3/library/socket.html#socket.socket.connect). Also note that the server does not [sendall()](https://docs.python.org/3/library/socket.html" \l "socket.socket.sendall" \o "socket.socket.sendall)/[recv()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recv" \o "socket.socket.recv) on the socket it is listening on but on the new socket returned by [accept()](https://docs.python.org/3/library/socket.html#socket.socket.accept).

The first two examples support IPv4 only.

Copy

*# Echo server program*

**import** **socket**

HOST = '' *# Symbolic name meaning all available interfaces*

PORT = 50007 *# Arbitrary non-privileged port*

**with** socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) **as** s:

s.bind((HOST, PORT))

s.listen(1)

conn, addr = s.accept()

**with** conn:

print('Connected by', addr)

**while** **True**:

data = conn.recv(1024)

**if** **not** data: **break**

conn.sendall(data)

Copy

*# Echo client program*

**import** **socket**

HOST = 'daring.cwi.nl' *# The remote host*

PORT = 50007 *# The same port as used by the server*

**with** socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) **as** s:

s.connect((HOST, PORT))

s.sendall(b'Hello, world')

data = s.recv(1024)

print('Received', repr(data))

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to all the addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

Copy

*# Echo server program*

**import** **socket**

**import** **sys**

HOST = **None** *# Symbolic name meaning all available interfaces*

PORT = 50007 *# Arbitrary non-privileged port*

s = **None**

**for** res **in** socket.getaddrinfo(HOST, PORT, socket.AF\_UNSPEC,

socket.SOCK\_STREAM, 0, socket.AI\_PASSIVE):

af, socktype, proto, canonname, sa = res

**try**:

s = socket.socket(af, socktype, proto)

**except** **OSError** **as** msg:

s = **None**

**continue**

**try**:

s.bind(sa)

s.listen(1)

**except** **OSError** **as** msg:

s.close()

s = **None**

**continue**

**break**

**if** s **is** **None**:

print('could not open socket')

sys.exit(1)

conn, addr = s.accept()

**with** conn:

print('Connected by', addr)

**while** **True**:

data = conn.recv(1024)

**if** **not** data: **break**

conn.send(data)

Copy

*# Echo client program*

**import** **socket**

**import** **sys**

HOST = 'daring.cwi.nl' *# The remote host*

PORT = 50007 *# The same port as used by the server*

s = **None**

**for** res **in** socket.getaddrinfo(HOST, PORT, socket.AF\_UNSPEC, socket.SOCK\_STREAM):

af, socktype, proto, canonname, sa = res

**try**:

s = socket.socket(af, socktype, proto)

**except** **OSError** **as** msg:

s = **None**

**continue**

**try**:

s.connect(sa)

**except** **OSError** **as** msg:

s.close()

s = **None**

**continue**

**break**

**if** s **is** **None**:

print('could not open socket')

sys.exit(1)

**with** s:

s.sendall(b'Hello, world')

data = s.recv(1024)

print('Received', repr(data))

The next example shows how to write a very simple network sniffer with raw sockets on Windows. The example requires administrator privileges to modify the interface:

Copy

**import** **socket**

*# the public network interface*

HOST = socket.gethostbyname(socket.gethostname())

*# create a raw socket and bind it to the public interface*

s = socket.socket(socket.AF\_INET, socket.SOCK\_RAW, socket.IPPROTO\_IP)

s.bind((HOST, 0))

*# Include IP headers*

s.setsockopt(socket.IPPROTO\_IP, socket.IP\_HDRINCL, 1)

*# receive all packets*

s.ioctl(socket.SIO\_RCVALL, socket.RCVALL\_ON)

*# receive a packet*

print(s.recvfrom(65565))

*# disabled promiscuous mode*

s.ioctl(socket.SIO\_RCVALL, socket.RCVALL\_OFF)

The next example shows how to use the socket interface to communicate to a CAN network using the raw socket protocol. To use CAN with the broadcast manager protocol instead, open a socket with:

Copy

socket.socket(socket.AF\_CAN, socket.SOCK\_DGRAM, socket.CAN\_BCM)

After binding (CAN\_RAW) or connecting ([CAN\_BCM](https://docs.python.org/3/library/socket.html#socket.CAN_BCM)) the socket, you can use the [socket.send()](https://docs.python.org/3/library/socket.html" \l "socket.socket.send" \o "socket.socket.send) and [socket.recv()](https://docs.python.org/3/library/socket.html" \l "socket.socket.recv" \o "socket.socket.recv) operations (and their counterparts) on the socket object as usual.

This last example might require special privileges:

Copy

**import** **socket**

**import** **struct**

*# CAN frame packing/unpacking (see 'struct can\_frame' in <linux/can.h>)*

can\_frame\_fmt = "=IB3x8s"

can\_frame\_size = struct.calcsize(can\_frame\_fmt)

**def** build\_can\_frame(can\_id, data):

can\_dlc = len(data)

data = data.ljust(8, b'**\x00**')

**return** struct.pack(can\_frame\_fmt, can\_id, can\_dlc, data)

**def** dissect\_can\_frame(frame):

can\_id, can\_dlc, data = struct.unpack(can\_frame\_fmt, frame)

**return** (can\_id, can\_dlc, data[:can\_dlc])

*# create a raw socket and bind it to the 'vcan0' interface*

s = socket.socket(socket.AF\_CAN, socket.SOCK\_RAW, socket.CAN\_RAW)

s.bind(('vcan0',))

**while** **True**:

cf, addr = s.recvfrom(can\_frame\_size)

print('Received: can\_id=**%x**, can\_dlc=**%x**, data=**%s**' % dissect\_can\_frame(cf))

**try**:

s.send(cf)

**except** **OSError**:

print('Error sending CAN frame')

**try**:

s.send(build\_can\_frame(0x01, b'**\x01\x02\x03**'))

**except** **OSError**:

print('Error sending CAN frame')

Running an example several times with too small delay between executions, could lead to this error:

Copy

**OSError**: [Errno 98] Address already **in** use

This is because the previous execution has left the socket in a TIME\_WAIT state, and can’t be immediately reused.

There is a [socket](https://docs.python.org/3/library/socket.html#module-socket) flag to set, in order to prevent this, socket.SO\_REUSEADDR:

Copy

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

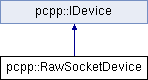
s.bind((HOST, PORT))

the SO\_REUSEADDR flag tells the kernel to reuse a local socket in TIME\_WAIT state, without waiting for its natural timeout to expire.

**pcpp::RawSocketDevice Class Reference**

#include <**[RawSocketDevice.h](https://pcapplusplus.github.io/v1904/Documentation/a00146_source.html)**>

Inheritance diagram for pcpp::RawSocketDevice:



|  |  |
| --- | --- |
| Public Types | |
| enum | [**RecvPacketResult**](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a06ec60dc8d38dfab0333fb793ad0da76) { **[RecvSuccess](https://pcapplusplus.github.io/v1904/Documentation/a01386.html" \l "a06ec60dc8d38dfab0333fb793ad0da76ad36c34a2ebc972da8d8ad02b1e1d8c70)** = 0, **[RecvTimeout](https://pcapplusplus.github.io/v1904/Documentation/a01386.html" \l "a06ec60dc8d38dfab0333fb793ad0da76a0ad5957368155861234acd2c5251b95c)** = 1, **[RecvWouldBlock](https://pcapplusplus.github.io/v1904/Documentation/a01386.html" \l "a06ec60dc8d38dfab0333fb793ad0da76abaedaa450ddecac58433cf954fdfd8b8)** = 2, **[RecvError](https://pcapplusplus.github.io/v1904/Documentation/a01386.html" \l "a06ec60dc8d38dfab0333fb793ad0da76adce5f5b2a3d83a5816da7520ee4228ea)** = 3 } |
|  | |

|  |  |
| --- | --- |
| Public Member Functions | |
|  | [**~RawSocketDevice**](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a16669eaf3e120d99ff07a07223aca9b6) () |
|  | |
| [**RecvPacketResult**](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a06ec60dc8d38dfab0333fb793ad0da76) | [**receivePacket**](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a184663a2718b6e4f5f7ad353d840dfbc) (**[RawPacket](https://pcapplusplus.github.io/v1904/Documentation/a01382.html)** &rawPacket, bool blocking=true, int timeout=-1) |
|  | |
| int | [**receivePackets**](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a83f604ddccbb3dad4dbee74bcda6d21d) (**[RawPacketVector](https://pcapplusplus.github.io/v1904/Documentation/a00202.html" \l "a4c0775fb93a0867d2afa7992d55957f0)** &packetVec, int timeout, int &failedRecv) |
|  | |
| bool | [**sendPacket**](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a3a7822e719e2f5895b3bee1523327e08) (const **[RawPacket](https://pcapplusplus.github.io/v1904/Documentation/a01382.html)** \*rawPacket) |
|  | |
| int | [**sendPackets**](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a348a0cfddc2d9b04857fd8bf8402b6ba) (const **[RawPacketVector](https://pcapplusplus.github.io/v1904/Documentation/a00202.html" \l "a4c0775fb93a0867d2afa7992d55957f0)** &packetVec) |
|  | |
| virtual bool | [**open**](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#afadbe3e75178913f1d9c2cc3f6d02909) () |
|  | |
| virtual void | [**close**](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a040fdad86cc60ef3b289834251a942c5) () |
|  | |
| **- Public Member Functions inherited from [pcpp::IDevice](https://pcapplusplus.github.io/v1904/Documentation/a00758.html)** | |

Detailed Description

A class that wraps the raw socket functionality. A raw socket is a network socket that allows direct sending and receiving of IP packets without any protocol-specific transport layer formatting (taken from Wikipedia: <https://en.wikipedia.org/wiki/Network_socket#Raw_socket>). This wrapper class enables creation of a raw socket, binding it to a network interface, and then receiving and sending packets on it. Current implementation supports only Windows and Linux because other platforms provide poor support for raw sockets making them practically unusable. There are also major differences between Linux and Windows in raw socket implementation, let's mention some of the:

* On Windows administrative privileges are required for raw sockets creation, meaning the process running the code has to have these privileges. In Linux 'sudo' is required
* On Windows raw sockets are implemented in L3, meaning the L2 (Ethernet) layer is omitted by the socket and only L3 and up are visible to the user. On Linux raw sockets are implemented on L2, meaning all layers (including the Ethernet data) are visible to the user.
* On Windows sending packets is not supported, a raw socket can only receive packets. On Linux both send and receive are supported
* Linux doesn't require binding to a specific network interface for receiving packets, but it does require binding for sending packets. Windows requires binding for receiving packets. For the sake of keeping a unified and simple cross-platform interface this class requires binding for both Linux and Windows, on both send and receive

More details about opening the raw socket, receiving and sending packets are explained in the corresponding class methods. Raw sockets are supported for both IPv4 and IPv6, so you can create and bind raw sockets to each of the two. Also, there is no limit on the number of sockets opened for a specific IP address or network interface, so you can create multiple instances of this class and bind all of them to the same interface and IP address.

Member Enumeration Documentation

[◆](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a06ec60dc8d38dfab0333fb793ad0da76)RecvPacketResult

|  |
| --- |
| enum **[pcpp::RawSocketDevice::RecvPacketResult](https://pcapplusplus.github.io/v1904/Documentation/a01386.html" \l "a06ec60dc8d38dfab0333fb793ad0da76)** |

An enum for reporting packet receive results

|  |  |
| --- | --- |
| Enumerator | |
| RecvSuccess | Receive success |
| RecvTimeout | Receive timeout - timeout expired without any packets being captured |
| RecvWouldBlock | Receive would block - in non-blocking mode if there are no packets in the rx queue the receive method will return immediately with this return value |
| RecvError | Receive error, usually will be followed by an error log |

Constructor & Destructor Documentation

[◆](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a16669eaf3e120d99ff07a07223aca9b6)~RawSocketDevice()

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pcpp::RawSocketDevice::~RawSocketDevice | ( |  | ) |  |

A d'tor for this class. It closes the raw socket if not previously closed by calling [**close()**](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a040fdad86cc60ef3b289834251a942c5)

Member Function Documentation

[◆](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a040fdad86cc60ef3b289834251a942c5)close()

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | virtual void pcpp::RawSocketDevice::close | ( |  | ) |  | | virtual |

Close the raw socket

Implements **[pcpp::IDevice](https://pcapplusplus.github.io/v1904/Documentation/a00758.html" \l "acebd2ae24e39e620ed66f65e6623089b)**.

[◆](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#afadbe3e75178913f1d9c2cc3f6d02909)open()

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | virtual bool pcpp::RawSocketDevice::open | ( |  | ) |  | | virtual |

Open the device by creating a raw socket and binding it to the network interface specified in the c'tor

**Returns**

True if device was opened successfully, false otherwise with a corresponding error log message

Implements **[pcpp::IDevice](https://pcapplusplus.github.io/v1904/Documentation/a00758.html" \l "ad7bed3f1134e8b4329295b4af6733b9f)**.

[◆](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a184663a2718b6e4f5f7ad353d840dfbc)receivePacket()

|  |  |  |  |
| --- | --- | --- | --- |
| [**RecvPacketResult**](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a06ec60dc8d38dfab0333fb793ad0da76) pcpp::RawSocketDevice::receivePacket | ( | [**RawPacket**](https://pcapplusplus.github.io/v1904/Documentation/a01382.html) & | rawPacket, |
|  |  | bool | blocking = true, |
|  |  | int | timeout = -1 |
|  | ) |  |  |

Receive a packet on the raw socket. This method has several modes of operation:

* Blocking/non-blocking - in blocking mode the method will not return until a packet is received on the socket or until the timeout expires. In non-blocking mode it will return immediately and in case no packets are on the receive queue **[RawSocketDevice::RecvWouldBlock](https://pcapplusplus.github.io/v1904/Documentation/a01386.html" \l "a06ec60dc8d38dfab0333fb793ad0da76abaedaa450ddecac58433cf954fdfd8b8)** will be returned. Unless specified otherwise, the default value is blocking mode
* Receive timeout - in blocking mode, the user can set a timeout to wait until a packet is received. If the timeout expires and no packets were received, the method will return **[RawSocketDevice::RecvTimeout](https://pcapplusplus.github.io/v1904/Documentation/a01386.html" \l "a06ec60dc8d38dfab0333fb793ad0da76a0ad5957368155861234acd2c5251b95c)**. The default value is a negative value which means no timeout

There is a slight difference on this method's behavior between Windows and Linux around how packets are received. On Linux the received packet contains all layers starting from the L2 (Ethernet). However on Windows raw socket are integrated in L3 level so the received packet contains only L3 (IP) layer and up.

**Parameters**

|  |  |  |
| --- | --- | --- |
| [out] | **rawPacket** | An empty packet instance where the received packet data will be written to |
| [in] | **blocking** | Indicates whether to run in blocking or non-blocking mode. Default value is blocking |
| [in] | **timeout** | When in blocking mode, specifies the timeout [in seconds] to wait for a packet. If timeout expired and no packets were captured the method will return **[RawSocketDevice::RecvTimeout](https://pcapplusplus.github.io/v1904/Documentation/a01386.html" \l "a06ec60dc8d38dfab0333fb793ad0da76a0ad5957368155861234acd2c5251b95c)**. Zero or negative values mean no timeout. The default value is no timeout |

**Returns**

The method returns one on the following values:

* [**RawSocketDevice::RecvSuccess**](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a06ec60dc8d38dfab0333fb793ad0da76ad36c34a2ebc972da8d8ad02b1e1d8c70) is returned if a packet was received successfully
* [**RawSocketDevice::RecvTimeout**](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a06ec60dc8d38dfab0333fb793ad0da76a0ad5957368155861234acd2c5251b95c) is returned if in blocking mode and timeout expired
* [**RawSocketDevice::RecvWouldBlock**](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a06ec60dc8d38dfab0333fb793ad0da76abaedaa450ddecac58433cf954fdfd8b8) is returned if in non-blocking mode and no packets were captured
* [**RawSocketDevice::RecvError**](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a06ec60dc8d38dfab0333fb793ad0da76adce5f5b2a3d83a5816da7520ee4228ea) is returned if an error occurred such as device is not opened or the recv operation returned some error. A log message will be followed specifying the error and error code

[◆](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a83f604ddccbb3dad4dbee74bcda6d21d)receivePackets()

|  |  |  |  |
| --- | --- | --- | --- |
| int pcpp::RawSocketDevice::receivePackets | ( | [**RawPacketVector**](https://pcapplusplus.github.io/v1904/Documentation/a00202.html#a4c0775fb93a0867d2afa7992d55957f0) & | packetVec, |
|  |  | int | timeout, |
|  |  | int & | failedRecv |
|  | ) |  |  |

Receive packets into a packet vector for a certain amount of time. This method starts a timer and invokes the **[receivePacket()](https://pcapplusplus.github.io/v1904/Documentation/a01386.html" \l "a184663a2718b6e4f5f7ad353d840dfbc)** method in blocking mode repeatedly until the timeout expires. All packets received successfully are put into a packet vector

**Parameters**

|  |  |  |
| --- | --- | --- |
| [out] | **packetVec** | The packet vector to add the received packet to |
| [in] | **timeout** | Timeout in seconds to receive packets on the raw socket |
| [out] | **failedRecv** | Number of receive attempts that failed |

**Returns**

The number of packets received successfully

[◆](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a3a7822e719e2f5895b3bee1523327e08)sendPacket()

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| bool pcpp::RawSocketDevice::sendPacket | ( | const **[RawPacket](https://pcapplusplus.github.io/v1904/Documentation/a01382.html)** \* | rawPacket | ) |  |

Send an Ethernet packet to the network. L2 protocols other than Ethernet are not supported in raw sockets. The entire packet is sent as is, including the original Ethernet and IP data. This method is only supported in Linux as Windows doesn't allow sending packets from raw sockets. Using it from other platforms will also return "false" with a corresponding error log message

**Parameters**

|  |  |  |
| --- | --- | --- |
| [in] | **rawPacket** | The packet to send |

**Returns**

True if packet was sent successfully or false if the socket is not open, if the packet is not Ethernet or if there was a failure sending the packet

[◆](https://pcapplusplus.github.io/v1904/Documentation/a01386.html#a348a0cfddc2d9b04857fd8bf8402b6ba)sendPackets()

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| int pcpp::RawSocketDevice::sendPackets | ( | const **[RawPacketVector](https://pcapplusplus.github.io/v1904/Documentation/a00202.html" \l "a4c0775fb93a0867d2afa7992d55957f0)** & | packetVec | ) |  |

Send a set of Ethernet packets to the network. L2 protocols other than Ethernet are not supported by raw sockets. The entire packet is sent as is, including the original Ethernet and IP data. This method is only supported in Linux as Windows doesn't allow sending packets from raw sockets. Using it from other platforms will return "false" with an appropriate error log message

**Parameters**

|  |  |  |
| --- | --- | --- |
| [in] | **packetVec** | The set of packets to send |

**Returns**

The number of packets sent successfully. For packets that weren't sent successfully there will be a corresponding error message printed to log